A Literature Review of Spreadsheet Technology

Alexander Asp Bock

IT University Technical Report Series **TR-2016-199**

ISSN 1600–6100 **November 2016**

ISSN 1600–6100

ISBN 978-87-7949-364-3

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: **+45 72 18 50 00**
Telefax: **+45 72 18 50 01**
Web **www.itu.dk**

A Literature Review of Spreadsheet Technology

Alexander Asp Bock*

Software and Systems, IT University of Copenhagen

November 2016

Abstract

It was estimated that there would be over 55 million end-user programmers in 2012 [1] in many different fields such as engineering, insurance and banking, and the numbers are not expected to have dwindled since. Consequently, technological advancements of spreadsheets is of great interest to a wide number of people from different backgrounds. This literature review presents an overview of research on spreadsheet technology, its challenges and its solutions. We also attempt to identify why software developers generally frown upon spreadsheets and how spreadsheet research can help alter this view.

Contents

1 Introduction

Spreadsheets have existed since the 1970's [2, 3, 4] and many commercial and research spreadsheet applications are in widespread use today: LibreOffice's open-source OfficeCalc [5], Gnumeric [6], Apple's Numbers [7], VisiCalc [4, 8], Lotus 1–2–3 [4] and Google spreadsheets [9]. The most popular commercial application is undoubtedly Microsoft Excel. Spreadsheet users are so-called end-users or end-user programmers, people who are not trained IT professionals, but use programs such as spreadsheets as a means to an end, and are often domain experts in their respective fields. These communities include, but are by no means limited to, scientific, financial, engineering and governmental domains [10].

*E-mail: albo@itu.dk

Spreadsheets have also been used for educational purposes such as an introductory course in programming [11, see section 6] [12, 13]. They vastly outnumber the number of professional programmers, and so spreadsheet end-user programming can be viewed as a highly ubiquitous form of programming. As a result, the spreadsheet research community is highly active and largely agrees on the popularity and abundance of spreadsheets as valuable organisational tools [14].

Despite their abundance, there seems to be a tendency from the software development community to frown upon spreadsheet programming as not being "real programming" [15, 16, 17, 18]. Casimir even suggests that spreadsheets are just plain boring for programmers [17].

Spreadsheets can generally be described as first-order, declarative and functional languages with a visual interface. Compared to traditional programs they do not have a compilation step, but instead an *edit-run* work cycle [14]. Another distinctive feature of spreadsheets is their support for *automatic recalculation* where cell values are instantly updated in response to user modifications.

This study explores the current challenges in spreadsheet technology and consolidates, compares and critiques state-of-the-art approaches to solving them. More specifically, we aim to answer the following questions:

1. What are some notable advances in spreadsheet technology?
2. How do these advances affect end-user development?
3. Do professionals in the software industry not consider spreadsheet programming as "real" programming, and if so, why?

The contributions of this paper are a broad coverage and critical discussion of different spreadsheet technologies and their different foci. We stress that this literature review is not intended to cover *all* the current literature on spreadsheets, but to provide a general overview of the research on spreadsheets for interested readers. We have chosen to divide the literature in this study into several categories based on which part of the spreadsheet paradigm they address. In each section, we succinctly highlight the research contributions, results and other relevant points and compare the approaches to those from other papers if applicable. For the reader's convenience, a table is available in the appendix that gives an overview of the referenced literature and to which categories they belong.

We conclude with a small table of contents.

1.1 Spreadsheet Terminology

In this section, we provide a short introduction to some of the general spreadsheet terminology used throughout this paper.

A spreadsheet is a graphical user interface tool that consists of a number of *worksheets* each of which is composed of a rectangular grid of *cells*. Each cell can contain different values such as text, a number, a date or a *formula* to name a few. A cell containing a formula can refer to another cell by way of references. Rows and columns in the cell grid each have a unique number and letter respectively. Numbers usually start at 1 and letters at `A`. A reference can be either absolute or relative. An absolute reference refers to a cell using its exact address such as `$B$10` (tenth row, second column). A relative reference instead contains offsets relative to the containing cell; the relative reference `-1, 5` in cell `$B$10` would refer to the cell at `$A$15`. Note that the type of a reference does not affect formula evaluation, but affects the formula when it is copied. There are different ways of displaying references. The R1C1 format lists rows followed by columns. For instance, `R2C1` is an absolute reference to the cell in the second row and first column and `R[-1]C[-1]` is a relative reference to a cell located one row and column above and to the left of the cell containing the reference. In the A1 format, the columns are listed first. Absolute references are prefixed with a dollar sign and relative references omit it, thus relative to cell `B2`, `A1` is a reference to the previous column and row, while `$B$10` is an absolute reference to column `B` and the tenth row. Both formats support combinations of absolute and relative rows and columns in a single reference along with cell ranges that refer to a cell area. Lastly, Funcalc [19] uses another format called the `CORO` format, which is zero-based and indexes with the column first, but is otherwise similar to the `R1C1` format. Zero-indexing is convenient for array access in most contemporary programming languages since they also use zero-based indices, so Funcalc uses this format internally.

Formulas are expressions that perform operations on cells to yield a result. A classical example is the `SUM` function: `SUM($A$1:$A$20)` sums the values in the cells spanned by the range `$A$1:$A$20`, i.e. the first twenty rows of the first column. Most modern spreadsheet applications support a wide range of these types of aggregation functions. They also generally support functions for calendars, currencies and databases etc.

2 Sheet Representation

Spreadsheets usually present the user with a rectangular 2D grid of cells. Each cell can contain values, formulas, graphs and even buttons and graphics [20]. Each cell can reference other cells through absolute or relative references and can thus be viewed as a global, virtual address space [18]. The spatial representation and internal memory layout is a fundamental and significant part of a spreadsheet program. It requires careful design in order to represent a multitude of cell layouts efficiently. Unfortunately, not much literature can be found on the subject. As an example of its importance, consider a contrived example where a user has inserted some data into the four extreme corners of a spreadsheet. A naive 2D block of cells would consume an unacceptable amount of memory to store only these four cells. Excel 2013 is capable of representing 16,384 columns and 1,048,576 rows [2], assuming each cell took up only 1 byte for simplicity, the naive approach would consume approximately 17GB of memory to represent these 4 cells, which is a mere 2.33E-10% of the total available space. Any cell representation scheme also affects the efficiency of recalculation. Assume the same four cells, that one of them is modified (triggering recalculation) and that there are no cyclic dependencies, the recalculation process would need to scan over 17 billion cells to update at most 3 cells. Surely, better alternatives must exist.

A more efficient strategy uses a quadtree [2], a spatial data structure that can efficiently represent sparse objects in 2D. A 2D space is recursively subdivided into four subquads. Each internal node has four children, one for each of the subquads. The idea is to recursively subdivide a large 2D space into four smaller, equally sized quads or rectangles, usually to a certain depth or dictated by some other criteria such as nesting depth. To query the quadtree with a 2D point, it is determined in which subquad the point should lie in based on its coordinates. This approach is recursively applied to subsequent subquads until either the point is found or an empty subquad is located, so the time complexity for querying is $O(\lg(N))$ where lg is the logarithmic function with base 4, i.e. $\log_4$. When the user creates, deletes, cuts or copies cells, the quadtree will need to be updated accordingly.

A quadtree is a good fit for dense and sparse data layouts, both of which should be accommodated by a spreadsheet application. Sestoft uses a modified, but similar, data structure dubbed QT4, represented by a quadruple-nested array of arrays of non-quadratic quads. Bit-shifting of coordinates are used to query the QT4 data structure and since its depth is fixed, access time is constant. Experiments have shown good performance results for various access patterns [2].

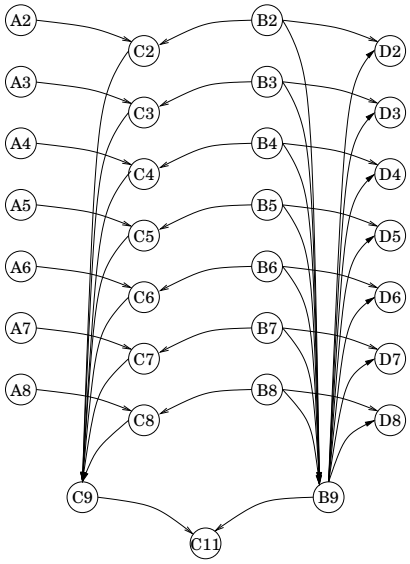
3 Recalculation

Automatic recalculation of cell values is a cornerstone of spreadsheet applications: When a user modifies a cell, all cells whose values depend upon it, are automatically updated accordingly. Therefore it is also vital to ensure that this process happens without noticeable delay to retain interactivity. The problem quickly becomes complex when we consider cell references and cell arrays, which can contain values, formulas, functions or even data that needs to be fetched from an external source.

As depicted in figure 1a most spreadsheet applications present users with a cell grid. Figure figure 1b shows dataflow between the same cells.

	A	B	C	D
1	Grade	Count	Product	Count %
2	-3	1	=A2*B2	=B2/$B$9*100
3	0	6	=A3*B3	=B3/$B$9*100
4	2	5	=A4*B4	=B4/$B$9*100
5	4	9	=A5*B5	=B5/$B$9*100
6	7	19	=A6*B6	=B6/$B$9*100
7	10	14	=A7*B7	=B7/$B$9*100
8	12	4	=A8*B8	=B8/$B$9*100
9	Sum	=SUM(B2:B8)	=SUM(C2:C8)	
10				
11		Average	=C9/B9	

(a)



(b)

Figure 1: A spreadsheet (figure 1a) and its corresponding support graph (figure 1b) [2].

Such a graph is used in many spreadsheet applications to track dependencies between non-empty cells and efficiently recalculate those affected by an update. One such graph is the support graph defined by Sestoft [2]. In this graph, there is an edge in the graph from a source cell to a target cell if the target cell depends on the source cell, and the source cell is said to *support* its target cells, hence the name. Because the edges point in the inverse direction of data dependencies, the support graph is analogous to a dataflow graph [21] where data flows along its edges. Conversely, the *dependency graph* is the inverse graph of the support graph where edges signify cell dependencies. The nomenclature for these graphs is not standardised, so we define exact terms for them here to avoid confusion. For example, Hoon et al. use *used-by* and *uses* respectively [22]. We choose to use the terms *support graph* and *dependency graph* since they most accurately capture their intent, and since the latter is used extensively in the literature on graphs and should thus be more immediate to a larger audience.

Before discussing some approaches to recalculation, we explain the difference between *static* and *dynamic* cell reference cycles [2]. When recalculating cells, cyclic dependencies are problematic. In Excel, the offending cells are marked by a `#CYCLE!` error value. If one cell reference, perhaps transitively, refers to itself, a static cycle has been found. On the other hand, a dynamic cycle occurs when cells contain non-strict expressions such as the `IF` function [2]. For non-strict functions, not all arguments need to be evaluated in order to fully evaluate the function. Consequently, dynamic cycles may or may not be discovered during recalculation. A spreadsheet can have both types of cycles, and a spreadsheet with a dynamic cycle also has a static cycle, but the converse is not true.

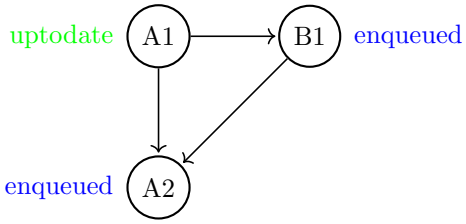
Sestoft [2] proposes two different strategies for recalculation which use the support graph. We will only discuss the one which is currently used in the Funcalc research spreadsheet application [19], called *standard minimal recalculation*.

The recalculation algorithm is *minimal* in the sense that it will visit each cell affected by a user update at most once. Barring the discovery of cycles, recalculation will make sure that the values of all cells are updated after its completion. Each cell can be in any of four states: *dirty* because it has not been computed yet, *computing* because its value is currently being computed, *enqueued* because the cell is currently waiting to be picked for computation on a queue, or *uptodate* if the cell has been updated. Recalculation is divided into two phases: **Mark** and **Evaluate** as described below. The algorithm starts out by identifying the *recalculation roots*, the set of cells that have been modified and all volatile cells that need to be recalculated, and putting them in a recalculation queue. It then makes sure that all cells affected by these roots are marked as *dirty*. This is the **Mark** phase. In the **Evaluate** phase, a cell is picked from the queue. If it is not already *uptodate* or *computing* (which would mean that there is a cycle), it is either *dirty* or *enqueued*, so it is marked as *computing* and evaluated. The evaluation of formulas lead to the evaluation of the cells that the formula expression refers to by way of recursive function calls. When a cell has been fully evaluated, it is marked as *uptodate* and its set of supported cells that are marked *dirty* are added to the queue. All these cells are marked as *enqueued* before they are added to the queue. If a cell is discovered that is in state *computing*, then a cycle has been found. The use of the *enqueued* state means that the recalculation can distinguish between cells that are waiting on the queue to be computed and cells

which have already been picked from the queue and are currently being computed. If we did not use separate states but used *computing* for both of them, cells could be put on the queue that would have that state and then be reported as wrongly causing a cycle when they were picked off of the queue. We illustrate this fact in figure 2. Imagine that a user changed the constant in cell `A1` from 42 to 43, so `A1` is the only recalculation root. All cells are initially marked as dirty. Cell `A1` will be marked as *uptodate* as it is a constant and cells `A2` and `B1` will be added to the recalculation queue. We would now normally have the scenario depicted in figure 2b. Let cell `B1` be the next cell to be picked from the queue. In this case the system would report a cycle because after cell `B1` has been fully evaluated it will add its set of supported cells to the recalculation queue and marking each of them as *computing* instead of *enqueued*. Then when cell `A2` is picked off of the queue and its state is found to be *computing*, the system will erroneously believe that it has found a cycle. Thus using *enqueued* allows us to differentiate between cells that are on the queue and those that are not.

	A	B
1	43	=A1+1
2	=A1+B1	

(a) A spreadsheet with some simple dependencies. Cell `A1` supports both cells `A2` and `B1`, while cell `B1` also supports cell `A2`.



(b) The corresponding support graph for the spreadsheet in figure 2a.

Figure 2: A spreadsheet and its support graph.

Sestoft also proposes an approximate topological sort on the order of cell dependencies to avoid deep recursive calls which may exceed the stack depth. This approach is also suggested in [23].

Serek and Poulsen examined optimised recalculation using the support graph in Funcalc [24]. Flemberg and Larsen further improved this algorithm in their Master thesis [25]. We discuss their work in section 3.2.

In [22], Hoon et al. implement a spreadsheet application using the pure and lazy functional language Clean. This has an impact on recalculation, because the spreadsheet is evaluated in a lazy fashion. Cell updates that are not visible to the user, because they are outside the view port, need not be re-evaluated until they come into view. This recalculation strategy is especially efficient if the user stays within the current area in the spreadsheet and is modifying a particular cell multiple times before moving the screen, because resources are not expended trying to evaluate dependencies of that cell until they become visible. Using Sestoft's strategy, recalculation of affected cells is done regardless of whether or not they are visible, wasting potentially unneeded computations. In contrast, lazy evaluation may affect moving around in the spreadsheet if a user moves into an area with lot of dependencies whose recalculation has been delayed. However, this could be done asynchronously to avoid affecting the user interface, and cells whose values have not yet been computed could be assigned the value `#GETTING_DATA` which is used by Excel for long-running calculations. The loading could also be performed speculatively and asynchronously by pre-computing some of the cells that surround the user's current viewport under the assumption that users will eventually move to some other part of the spreadsheet. More work is necessary to fully understand and compare these two approaches.

3.1 Dataflow

Dataflow will be a recurring term in this study, and so requires a more precise definition since it has multiple interpretations in the literature.

Fine-grained dataflow is implemented in hardware and dates back to the 1970's at MIT, with active researchers like K. Arvind and R. S. Nikhil. It was conceived in an attempt to design an entirely new hardware model that could rival the Von Neumann machine model and avoid the so-called *Von Neumann bottleneck* [26, 27]. Arvind and Nikhil developed the parallel language `Id` that was compiled into a paral-

lel machine language that used dataflow graphs to model the flow of computations and their dependencies [27]. The instructions were then executed on a special *Tagged Token Dataflow Architecture* (TTDA) with data-driven instruction scheduling based on the data dependencies of the dataflow graph. Computations had to wait for their dependencies to be computed, but otherwise independent computations could execute in parallel. Another example are also so-called asynchronous or self-timed circuits [28]. While not specifically associated with fine-grained dataflow, they exhibit similar properties. These circuits are not governed by a central clock as in contemporary CPUs, but instead use signals to communicate, akin to the flow of data that triggers execution of instructions in the `Id` language.

The language SISAL or *Streams and Iteration in a Single Assignment Language*, its accompanying compiler by Sarkar [29], and the works of John Hennessy and David Cann [30, 31, 32, 33] are examples of *coarse-grained* dataflow, and was implemented in software. This difference in granularity led to their respective names, since software can be considered coarse-grained compared to a more low-level, fine-grained hardware- and instruction-level approach. Coarse-grained dataflow programming was at its peak in the mid 1980's, but later declined due to the scarcity and cost of shared-memory multi-core processors, that are much more abundant today and exist on smaller scales such as personal laptops and smartphones.

The final type of dataflow is called *synchronous* dataflow programming. Gilles Kahn investigated this concept in 1974 with a formalisation of the semantics of a simplified language for communicating parallel processes modelled by a schema or network, that closely resembles a dataflow graph [34]. `Lustre` was designed for reactive systems in 1991 by Halbwachs et al. [35]. The authors define a dataflow model, and a language which augments this model with a concept of time-dependent flows, clocks and operators for construction of time-sensitive and event-driven programs. The synchronous aspect of `Lustre` is the formulation of conditions and relations using the semantics of the language that control the interplay of events.

3.2 Parallel Recalculation

Parallel programming has received renewed interest from the computer science research community in the last decade. A widespread manifestation of parallel machines is the *shared-memory multi-core* system where processors share a global address space, used for both storage and communication between parallel threads and processes. Today such systems can be found in anything from desktop and laptop computers to graphics processing units and mobile phones. However, leveraging this performance requires expert knowledge and experience to deliver performing and scalable solutions, something that end-users are rarely equipped to do. Systems that automatically use the available resources and accelerate end-user software are thus very attractive. This is further emphasised because real-world spreadsheets can easily become huge and overwhelming and contain complex formulas that can take a long time to recalculate. Some spreadsheets applications thus opt to allow the user to temporarily disable automatic recalculation during the development of spreadsheets, and later enable it when the spreadsheet's design and layout are satisfactory.

Contrary to what we expected, we did not find significant literature about parallel spreadsheets. We did discover a few patents nonetheless, but most of them are described in a vague or slightly obfuscated manner.

Previously, in section 3, we discussed different recalculation strategies and the dataflow and dependency graphs and mentioned how the graphs are suitable for parallel execution. Given this fact and the prevalence of contemporary multi-core systems, it is perhaps surprising that few attempts have been made to enable parallel recalculation. This is possibly either a testament to the difficulties in creating a satisfactory solution or that the subject is just not as interesting as the other aspects of spreadsheets.

Flemberg and Larsen [25] attempted to accelerate spreadsheet recalculation in Funcalc [19] in their Master thesis based on Sarkar's work on an optimising compiler for the SISAL language [36]. Sarkar developed algorithms for partitioning and scheduling dataflow computations which he claimed could be used for SISAL and any parallel language. Flemberg and Larsen suggest an improved heuristic for the partitioning algorithm

and provide a new scheduling algorithm using the Task Parallel Library (TPL) [37]. They do not support sheet-defined or volatile functions (see section 5.2 for the latter) and support a subset of the built-in functions, in order to test the general applicability of their partitioning algorithm. They use the TPL's dataflow constructs to create a dataflow pipeline for the partitioned tasks. They do not perform any benchmarks due to the effort of engineering required to evaluate performance and the fact that any speedup could be attributed to either partitioning, scheduling or both. The results indicated that their new heuristic is better suited for partitioning spreadsheet computations than Sarkar's original heuristic. Furthermore, all the requirements were fulfilled in all three example spreadsheets, created by the authors themselves.

Distributed systems for accelerated evaluation of spreadsheet computations have been explored by Abramson et al. and Nadiminti et al. [38, 39]. Abramson et al. suggest a distributed solution where an external tool is in charge of scheduling computations. The results are then read back into the spreadsheet. They focus mainly on simulations in Excel. Their system, ActiveSheets, consists of an Excel front-end and a back-end that automates the entire process using Nimrod, a research tool [40] which has a commercial counterpart, EnFuzion [41]. The parallel evaluation of the spreadsheet is based on the flow of data between cells, and is thus an example of coarse-grained dataflow (see section 3.1). Independent computations are sent to Nimrod and executed in parallel. Their system exploits both inter- and intra-cell parallelism. The authors do not provide any details to the actual implementation of ActiveSheets, but provide two case studies.

Nadiminti et al. [39] developed the Excel plugin ExcelGrid, and similar distributed systems that use grid systems, using .NET and a service oriented architecture (SOA). The system was designed to run multiple instances of the same task using different parameters, similar to a distributed SIMD (Single Instruction, Multiple Data) model. The user activates ExcelGrid through a graphical user interface, selecting input and output cells and the system collects input parameters from the spreadsheet entered by the user. The workload is distributed to a cluster of desktop computers that share a file system, then subsequently collected and returned via callbacks. The system can run on both enterprise-local networks and global networks. The authors conduct two experiments using different grid systems, but disclose few details about them, such as the size of their experiments.

Commercial applications also exist. SpreadsheetGear [42] is a collection of commercial plugins for Excel, one of which is a calculation engine that boasts multi-threaded recalculation. HPC Services for Excel developed by Microsoft enables workbooks and user-defined functions (those defined in Visual Basic, not sheet-defined functions, see section 5 for a discussion) to be offloaded to run on clusters of compute nodes via a SOA interface [43].

3.3 Templates

Also known as models, templates are minimal or compressed spreadsheets that specify a predefined layout to be used in development. Templates have to be instantiated into actual spreadsheets in which development can take place. Spreadsheet templates provide several benefits. First, the template is usually compressed such that repeated columns and rows of similar data are collapsed into a single one. For instance, some columns might contain the same formulas and can thus be collapsed into one. Second, they can be created by a domain expert and used by novice users, reducing errors and deviations. Finally, if the template is correct and error-free, so are any spreadsheet instantiations [44, 45]. Templates are thus relevant for our continued discussion on error handling in spreadsheets.

Abraham and Erwig [46] implemented an algorithm for inferring templates in. In earlier work, the Visual Template Specification Language or VISTL [47] and the Gencel [45] system were developed to specify, and generate spreadsheets from templates, respectively. A user specifies the spreadsheet template in VISTL and instantiates the template in Gencel according to his or her requirements. Gencel ensures that reference, range and type errors are not present such that the instantiated spreadsheet is also free from these errors, and that it abides by the original structure of the template. The intent of this work is to automate template creation by inferring a minimal, underlying template from a given

spreadsheet. The authors use the spatial analysis algorithms from UCheck (see section 4.3.3) to determine the overall layout of the current sheet. This avoids inferring a template from unrelated data in the same sheet. Next, sets of similar formulas are identified. The sets include as many of these as possible to generate a minimal template. Two formulas are deemed similar if they are *cp-similar*, i.e. if one could have resulted from a copy-paste of the other. Additionally, if the data is of the same type there is strong evidence that two formulas are cp-similar, otherwise the system has been made tolerant to minor data deviations in formulas. These sets, that are part of a similar pattern, are then overlaid to produce the template.

4 Bugs In Spreadsheets

In light of the ubiquity of spreadsheets, it is alarming that many studies have found that they often contain a large amount of errors [48, 49, 50], perhaps due to overconfidence in end-users' ability to avoid errors [48, 51, 52] or the trust that is put into computers to compute correct results [53]. We do not mean the built-in error codes such as `#N/A` or `#DIV/0!`, but bugs introduced by human error such as referring to an incorrect cell yielding an incorrect value in that cell. As observed by Powell et al. *No studies on spreadsheets themselves have shown errors to be rare or inconsequential* [50]. This is escalated by reports of losses of up to billions of dollars [54, 55, 56, 57], because spreadsheets are used for important business decisions. Consequently, a large part of the research on spreadsheets has been devoted to error detection and handling, as well as methods for visual feedback to report those errors in a manageable and understandable manner. For further reading on spreadsheet errors, Kruck [57] provides an overview of studies of spreadsheet errors, Powell et al. [50] conduct a critical review of the literature on spreadsheet errors in and Zhang et al. [58] evaluate the efficiency in spreadsheet anomaly detection of AmCheck [59], UCheck [11] and Dimension [60] in an empirical study.

There have been many suggestions for classifying spreadsheet errors. Powell et al. [50] summarise the requirements for a satisfactory taxonomy of error classification, and list some of the problems with existing classifications. We will not attempt to classify errors here. Instead we will categorise the relevant literature and divide them into the subsequent sections to reflect their approach to error detection and handling. These techniques include *smells* [10, 59, 61, 62], *data clone detection* and *reparation* [63], *type systems* [11, 14, 23, 64] and systems that use *units*, *labels* or *dimensions* in the spreadsheet [16, 60, 65, 66, 67, 68], and *debugging* and *testing* [21, 51, 52, 69, 70, 71].

4.1 Smells

Code smells are yet another concept borrowed from the world of software development. First identified by Fowler [72], they are a collection of indications of bad practices and patterns that may warrant a refactoring of code. They make the code "smell" wrong, hence the name. Research has attempted to translate existing code smells to spreadsheets or define new smells [10, 59, 61]. We will collectively refer to these as *spreadsheet smells*.

Dou et al. [59] created *AmCheck* for detecting and repairing *ambigous computation smells*, which occur when a group of cells have different computational semantics in their formulas. Such a group is called a *cell array* and is a frequent construct in spreadsheets. The smell can occur when a cell formula is copy-pasted to other cells, initially retaining its computational semantics, but later on modifications change the semantics. The authors define two types of smells: The *missing formula smell* occurs when some cells in a cell array are not defined by any formula, and the *inconsistent formula smell* occurs when cells have differing formulas. The system analyses the spreadsheet and tries to detect these types of errors using a two-step process: First, cell arrays are identified. This involves a preliminary step of finding isolated regions (*snippets* in the text) using the same approach suggested by Abraham et al. [11] using soft and hard fences (see section 4.3.3). Second, it tries to find a common pattern, and if that fails, the program synthesis method from [73] is employed to create a common pattern that satisfies the constraints imposed by the collective formulas of the cell array. The user is then presented with suggestions for correcting identified mistakes. Their experiments on the EUSES corpus [74] showed that

44.7% of the spreadsheets that contained cell arrays, also contained at least one smell (and 27.3% of spreadsheets with formulas). They also report some cases of false positives, but AmCheck allows users to reject such cases through manual inspection. They also report false negatives that are caused by cell arrays not being detected properly. AmCheck does not handle conditionals in formulas which was one source of the false negatives. This was left as future work. Dou et al. remark that some smells will go undetected using systems such as UCheck [11] that check consistency using user-defined labels, or systems that use dimension inference [60, 66, 67] if the smells are not part of the erroneous cells detected by those systems.

Zhang et al. [58] evaluate the precision, recall rate, efficiency and scope of AmCheck, UCheck [11] and Dimension [60], and found that AmCheck outperformed them both, although the other two systems could find different types of anomalies. Suggestions for improvements are given for all systems.

In later work, Dou et al. [75] created *CACheck* based on AmCheck, yielding higher precision in detecting smelly cell arrays. CACheck differs from AmCheck in the following ways: CACheck can detect inhomogeneous cell arrays. A cell array is homogeneous if all cell formulas in the array refer to cells in the same row/column for column- and row-based cell arrays respectively (a row-based cell array is a row of consecutive cells). On the other hand, a cell array is inhomogeneous if there exists a cell in the cell array whose formula does not abide by the criteria of homogeneity. New observations about cell arrays enable CACheck to identify cell arrays with higher precision. For instance, if two sets of cell arrays, one row-based and one column-based, can describe the same region, one is selected and the other removed from consideration. CACheck is evaluated on both the EUSES corpus [74] and the Enron corpus [76], and not just the former.

Dou et al. [61] also developed the TableCheck system for detecting clones of and smells in tables with the same or similar computational semantics. Here, tables refer to rectangular areas of cells and the smells refer to missing or inconsistent computations between table clones. This is much like the cell arrays defined by AmCheck and CACheck. More specifically, a table clone is a pair of tables whose corresponding cells share the same headers and have the same computational semantics. They are also required to have at least two rows and columns. We now give a terse description of the algorithm. TableCheck proceeds by first determining the types of all cells using the methods described in [77] and [78], it then infers table headers and creates a lookup table with headers as keys and cell references that are covered by the given header as values. Table headers are identified by scanning left and up, for row and column headers respectively, and finding the first label cell (a cell containing text barring cases such as error values like `#N/A` etc.). This lookup table is then used to identify table clones by creating groups of table clones that satisfy the requirements for tables being clones. Each group of table clones is then examined for smells such as the *missing formula* smell for cells that are missing formulas when other cells in the table have formulas, or the *inconsistent formula* smell for cells that have different computational semantics. The algorithm is also extended to suppress false positives. The authors state that TableCheck outperforms AmCheck [59], CACheck [75], CUSTODES [62], UCheck [11], Dimension [60] and Excel, detecting table clones and smells in the EUSES spreadsheet corpus [74] with 92.2% and 85.5% precision respectively, while those frameworks only achieve at most a 35.6% smell detection rate. Also, some of the aforementioned frameworks were built to detect different smells, for instance, UCheck was developed as a type checker for spreadsheets, not for table clone detection. TableCheck found that 21.8% (352 out of 1617 spreadsheets with formulas) in the EUSES corpus contain manually verified table clones.

Cheung et al. [62] developed the CUSTODES framework to detect smells using a clustering technique that groups cells together based on strong and weak features. Strong features are formulas and references between cells (dependencies), and their similarity with other cells. Weak features are things such as labels, layouts and fonts. Labels are also used in checking consistency in spreadsheets, which we will discuss further in section 4.3.3. CUSTODES works as follows: First, cell formulas are identified. Second, the identified formulas are then clustered based on their strong and weak features. Finally, a bootstrapping algorithm is applied to the results. Smells are detected by using a *local outlier factor*

or LOF for each clustered cell, that finds outliers using the density of the feature space of the cluster cells. LOF is based on the assumption that the density of an inlier should be similar to the density of its neighbours, while the density of an outlier is lower. The LOF score of each cluster cell is the ratio of the local density of the cell and the average local density of its neighbours. Based on this detection, the outliers are then categorised into four smell categories: *Missing formula smell*, *dissimilar reference smell*, *Dissimilar operation smell* and *Hard-coded constant smell*. Experiments showed that CUSTODES outperforms existing smell detection frameworks.

4.2 Data Clones

A data clone is created when users copy and paste cells that contain formulas. If the user wishes to maintain consistency between all data clones and their original, he or she needs to update all copies. This is problematic since it may not be clear where a clone originated. This is related to the *don't-repeat-yourself* or DRY principle from software engineering, where data clones are also used to identify repeated code. Detection of data clones can inform the user of their presence and he or she can take actions by e.g. employing alternative techniques to achieve a similar goal such as linking where data clones are automatically updated when their source changes.

Hermans et al. [63] developed a technique for detecting data clones in spreadsheets. They define a clone as a copy of a cell and a clone cluster as a collection of clones that have the same values as another cluster. *Near-miss* clone clusters are clusters where almost all values are the same, which can happen if a copy is modified, but the original cluster is not. They are useful for detecting data clones that should be equal, but are not, perhaps due to missing updates in the clone. The authors found that 86 of the spreadsheets in EUSES corpus [74] (out of more than 4000 spreadsheets) contained manually verified data clones (for sensible, minimal algorithmic parameters), yielding a precision of 54.8%. This suggests that data clones are somewhat prevalent in real-world spreadsheets (data clones were found in approximately 5% of the spreadsheets in the EUSES corpus that contain formulas).

In their algorithm, cells that contain numerical values or formulas are first identified (cells that contain string values are ignored since they are usually copied as labels, e.g. "Total" to denote a sum of values). A lookup table is created for all cell values with their values as keys and all their locations as values. Entries that occur only once as either a constant or a formula cannot have clones because they were never copied, and are thus removed from the table. All cells in the table are then clustered together with their neighbours if the neighbours also occur in the table and are all either constants or formula cells. The clustering technique should be capable of clustering irregular data layouts, similar to the CUSTODES framework in [62] that is also independent of existing layouts. However, the paper does not mention whether diagonal neighbours are considered, which would slightly limit its ability to detect irregular layouts. Whether this has any implications in practice is doubtful since cells are usually structured as rectangular areas consisting of rows and columns without meaningful diagonals. In the final step, formula clusters are compared to each constant cluster. Two clusters match if they contain the same values (and so are not *near-miss* clusters. The matching criteria can be configured to detect them). If the clusters differ in size, the smaller cluster's values must be a subset of the bigger cluster.

The algorithm finds some false positives due to unforeseen use of data clones: E.g. student grades are all low numbers that are repeated, and values can denote labels, such as years, which are also repeated as constants and in formulas across the spreadsheet. The authors visualise the data clones by generating a dataflow diagram (see section 6 for further details) and displaying pop-ups at each node of the diagram to distinguish a clone from an original. The visualisation is based on previous work in [10] and [78].

Two real-word case studies were made to uncover the implications of data clones. The authors found that the visualisation of data clones help users identify copies of data that they were not previously aware of, and that near-miss clones help indicate clones that should have been updated.

4.3 Type Systems

Type systems have long served to provide compile-time error detection of type misuse in programming languages, and today's programming languages have rich and powerful type systems. For example, it rarely makes sense to add a string to an integer. Type systems generally come in two forms: static and dynamic. Languages such as C/C++, C#, Java and Haskell are statically typed. Values must be explicitly annotated with their type (barring cases where a type inference system can infer its type from context) and types are determined at compile-time. Furthermore, a value with one type cannot be assigned to a variable with an incompatible type, although some systems relax this constraint e.g. in order to allow assignment of an integer to a variable of floating-point type. Languages such as Python, Ruby and Smalltalk are dynamically typed. Their types are determined and checked at runtime and do not need to be explicitly annotated in the code (there are mixtures such as Perl). Thus assignments between variables of different types are usually allowed. Languages like Scala and F# also boast a robust type inference system to alleviate the task of type annotation. Type systems could bestow the same benefits onto spreadsheets as they do for imperative and functional languages, especially considering that spreadsheets can be considered first-order, functional languages.

Most spreadsheet implementations use relaxed variations of a type system to differentiate between integers, strings, dates etc. Microsoft Excel does little type checking [14] which in some cases can result in unexpected errors. For instance in Microsoft Excel, some comparisons between an empty string and a number always results in `false` [23]. In Google spreadsheets, summing a range of cells containing numbers with one cell containing a string, silently ignores the string cell and sums the remaining values as shown in figure 3. Type checking could help avoid such (presumably accidental) situations. Whether this is actually an error depends on user perception and the design of the spreadsheet application, but in a programming language context this should definitely constitute an error.

	A
1	1
2	1
3	string
4	1
5	1
6	1
7	1
8	1
9	7

Figure 3: Cell range summation in Google spreadsheets. The cell range `A1:A8` is summed in cell `A9` using the `SUM` function. All cells except for cell `A3`, that contains a string, contain integers. Cell `A3` is silently ignored in the summation.

The introduction of such a system requires an investment from the end-users to learn how types and type inference work, and the cost of such a process is likely why type systems are not found as integrated systems in commercial spreadsheet applications [16]. As a result, many researchers tend to draw on the principles of these systems to implement their work, but keep the details transparent and provide explanatory and clear error messages in a format that is understandable for end-users.

4.3.1 Template Inference

Abramson and Erwig [14] have developed a type system and type inference algorithm for spreadsheets that is used to guide the inference of accurate spreadsheet templates created using the Visual Template Specification Language (VISTL) [47].

The type checking system is based on the notion of *cp-similarity* between cells. Two cells are considered *cp-similar* if the contents of one cell could have resulted from a copy-paste from the other cell, much like data clones. This is a likely scenario in spreadsheets, which support sophisticated copy operations. The authors introduce basic types and typing judgements for values, operations, cells, formulas, spreadsheets and templates, as well as a complete typing system. The

arguments for function types are explicitly given basic types to restrict functions to be first-order. Additionally, a *type expectation* is defined which maps cell addresses to constant types. Rows and columns in the template are then compressed if their types agree, in the same sense that a list of integers can be described as `[Int]` in functional languages. The authors use this definition to define spreadsheet types as a set of column (and row) types that each contain a sequence of cell types. Adjacent columns and rows of the same type can then be compressed.

The template inference algorithm proceeds by annotating all cells in a spreadsheet by an equivalence function a to generate an extended spreadsheet. Here, the equivalence function is the definition of *upstream type equivalent* between two cells, i.e. two cells are defined to be *upstream type equivalent* if they have the same cell types in a spreadsheet S. Since type-equivalence is a stronger condition than *cp-similarity*, type errors are detected which would not have been caught using the definition of *cp-similarity* alone. All columns are grouped according to this equivalence, and each group is maximally overlaid, i.e. grouped together as a single column in the resulting template. Lastly, the formulas are updated and the columns are shifted according to the maximum possible overlay (maximal according to the numbers of columns that can be overlaid). The process repeats until there are no more columns in S that can be overlaid and the algorithm is repeated for all rows in the spreadsheet. The result is a compressed version of the original spreadsheet S. A user can try to infer a template from a spreadsheet and the inference algorithm will then report any type errors that prohibit columns or rows from being overlaid.

We also mention ClassSheets by Engels et al. [79], which the authors describe as a more expressive form of spreadsheet specification. Instead of specifying templates in a visual language such as VISTL, it is specified in a diagram language that borrows concepts from the object-oriented *Unified Modeling Language* (UML). However, ClassSheets are automatically transformed into VISTL templates. Similar to a UML diagram, the template is specified by, possibly nested, relations that are annotated with names for identification and can include cells that they refer to. If a spreadsheet sums a list of income values, its template might be specified by a diagram called **Income** with a total of type `integer` and another diagram called **Item** with a one-to-many relation between them. **Item** also contains an `integer` called `value` that is summed in the **Income** diagram to compute the total income for a set of items. The sum can refer to the values of `Item` using dot notation: `Item.value`. Formal definitions of ClassSheets are also given in the paper and ensure that ClassSheets definitions are consistent and well-formed. The benefits of ClassSheets over VISTL templates include being more compositional in structure, explicit type annotations and reuse by referring to other diagrams. However, VISTL should be much more accessible for end-users than the UML-inspired layout of ClassSheets, especially with the type annotations.

4.3.2 Static Type Systems

Cheng et al. [23] devised a method for static analysis of spreadsheets and associated programs (macros or VBA programs) in the more classical sense. They define a core spreadsheet language, its syntax and semantics for the ensuing discussion. One such language expression is the evaluation of an entire spreadsheet, where the authors define a topological order similar to Sestoft [2]. Interestingly, they do not define the topological order as total to permit parallel computation of the ordering (see section 3.2 for a discussion of parallel recalculation strategies). The system ties abstract predicates to *zones* in an abstract domain, as defined in [64]. An abstract predicate is either a type or an abstract formula, while an abstract formula is defined as the effect on types in the spreadsheet which can be propagated through the spreadsheet e.g. `(`Z_1`, float)`, which denotes a sheet zone Z_1 with type `float`. Types are simplified like in [14]. For example, the type of the addition of two floats: `float + float` can be simplified to just `float`. Zones are thus compact representations of abstract formulas or types that carry type information through the spreadsheet, and are ultimately used in the static analysis. During static analysis, zones of equal type can be joined by a number of operators. For instance, two adjacent zones of type `int` can be joined into a single zone.

Cheng et al. implemented the static analysis in VBA and OCaml for Excel, and the tool was tested on the EUSES spreadsheet corpus

(spreadsheets that only contain constants were considered trivial and excluded in the experiments since types can be directly inferred for each cell) [74]. Their results indicate that their type system is reasonably effective at finding true positives in the EUSES corpus, with a manually verified detection rate of 30% (45 true positives out of 142 spreadsheets detected by the system). Some of the false positives were caused by intentional misuse of spreadsheet functions. The authors argue that these false positives would have been hard to detect in a testing framework, and so the number of false positives is acceptable. However, as the user would still need to manually verify each identified case, we disagree that this is acceptable. Furthermore, other type systems such as those that infer and use user-defined units and labels should also be able to catch these kinds of errors. We discuss these systems later in section 4.3.3.

Cheng et al. are the only authors that consider data validation. In spreadsheets, a cell can be augmented with data validation to ensure that users fill in a name that is a string or a number that is within a specific range. Such information is useful when devising a type system for spreadsheets. An interesting question is whether this would benefit other systems discussed in this section. They also find that their assumptions about the initial state of a spreadsheet and taking into account data verification information set by the user, faults are detected that would otherwise not be caught by testing frameworks such as those presented in section 4.4.

Lastly, we briefly mention a type system implemented by Florian Biermann in the Funcalc project [19]. As a spreadsheet is evaluated in the order described in section 3, the type system simply checks the types for all function applications and formulas, and raises appropriate errors if the expected types do not match. This system does not verify dynamic type errors that are only visible at runtime, e.g. an `IF` statement in a formula. The system is still in development at the time of writing and subject to change, so we will not discuss it further.

4.3.3 Units and Labels

As remarked by Erwig et al. in [16], a static typing may incur a high learning cost for end-users who are not acquainted with type systems. This higher cost is similar to what is described in the *attention investment* model discussed in [15], that models the likelihood of the investment of programmers in an activity based on the activity's expected payoff, cost and risk. Erwig et al. take a different approach to types where units are inferred from cells, that are then used to perform unit checking. A unit is simply an abstract concept such as fruit. A cell may denote the number of apples harvested by a company, although the type `apple` does not really exist in the spreadsheet. One could easily imagine a spreadsheet where a user attempts to add two different, abstract units. An ordinary type system would not be able to detect this, since the addition of "apples" and "oranges" involve the addition of two integers which would type check (unless of course units were embedded in the type systems as types or can be defined by the user as such [53]). The authors did not implement the system in this work, but discuss and flesh out the details for a fully fledged system. Their efforts are implemented in [11] which we discuss later.

Similar to the other literature discussed so far, Erwig et al. define a calculus to reason about spreadsheets, but in the context of units. Unit expressions are defined to allow for meaningful combinations of units: Dependent units and *and* and *or* units. In the context of the authors' fruit harvesting example, dependent units represent units such as `Fruit[Apple[Green]]` (or alternatively `Fruit[Apple][Green]`) because an apple is a fruit and has a colour. The colour is also dependent on the `Apple` unit as not all units may be green. The other two expressions allow the system to express units like `Fruit[Orange]&Month[May]` or `Fruit[Apple|Orange]`, which are both situations that can arise in two-dimensional tables, in this instance with fruits as columns and months as rows. They also define a context-free syntax, unit transformations for spreadsheet functions and operations, typing judgements, and unit simplification rules that allow complex units to be reduced to well-formed units. Failure to perform this reduction leads to a unit error in the spreadsheet which can be reported to the user. Although the authors avoid types entirely in the work described here, future work will attempt to consolidate their unit checking system with a type system by treating units as types.

Burnett and Erwig [65] implement the system

described above in later work. A *base layout rule* is given that is used to infer units based on the layout of cells. The system allows the user to visually reconfigure the inference if the inferred units are not satisfactory, by inserting lines that delimit cells and help the system correctly distinguish headers, borders and content. Borders are denoted by blank cells. Compared to the systems we have discussed thus far, this system is not fully automated and requires some user intervention to define and refine the logical rules that govern the unit inference. The authors also briefly discuss how the system could incorporate dimensions (such as meters per second etc.) in their system, which would need additional rules to infer division of e.g. meters with square meters. They also suggest a "unit view" for Excel, where the units for all cells are displayed instead of their cell contents. This is similar to the "formula view" where the formula definitions are displayed. This is a very interesting suggestion, in part because no other paper has suggested a similar extension to the author's knowledge. This would allow end-users to quickly get an overview of the inferred units to help define custom rules or understand and correct mistakes. The next unit system we discuss, will feature an automated unit inference system. As with any other non-automated system for end-user development, one is inclined to ask how much work must be done by the user, and if they are willing to invest resources into learning the system. No empirical studies were done in this work, but is left as future work.

In later work, Erwig et al. [11] define a new concept of proxy headers and implement a set of algorithms for header and unit inference. Their system is called UCheck. One benefit of automatic unit and header inference is that spreadsheets do not have to be manually annotated with units as some other work has done [53]. Nevertheless, in the first example given (using fruit harvesting), the `Fruit` unit is explicitly given as a unit in order to give the fruits in the spreadsheet a common ancestor. This is required since the system has no notion of the concept of fruits, and thus cannot infer this information itself, and so the base unit must be introduced somehow. A similar argument can be used for the months given in another example. Although the necessary amount of annotation is minimal in this example, the system still seems to require some annotation, and whether the amount of annotation will remain minimal in bigger, real-world spreadsheets is not known. The header inference algorithm uses a combination of cell classifications based on spatial and cell information. For example, a cell can be a header, a core cell that participates in some intermediate calculations, a footer that contains an aggregation formula or a filler cell used to separate tables. The system also uses cell distances and cell references to further classify cells. Configurable confidence levels are used to combine classification information from different algorithms. Headers are also assigned a level, since one header can have sub-headers. The inference algorithm also accounts for individual preference in header positioning. The system assumes sensible table layouts in spreadsheets. This is a fair assumption, but indicates that the system may fail in cases where an uncommon layout is used, but it would be very difficult to design a system that could account of all possible, arbitrary scenarios, so it makes sense for UCheck to focus on common layouts.

In the evaluation of their system, a comparison between UCheck and a static type system with user-defined annotations [53] is made based on a single case of unit error detection in only 10 spreadsheets, and the systems are deemed equally effective. In our opinion, this is a somewhat unfair comparison since the comparison is made for a relatively small set of spreadsheets. The authors also remark the lack of a representative set of spreadsheets is not available, though two years prior to their publication, the EUSES spreadsheet corpus was published [74] which many systems in this literature review has used in the evaluation of their work. The authors might not have been aware of this, but the set of spreadsheets used in their work (28 in total) is relatively small. They suggest an extension to the UCheck framework which can provide context-specific examples and change suggestions to the user. The latter is something that the GoalDebug system provides and has been developed by the same authors [69]. We will discuss GoalDebug in section 4.4. The authors argue that explicit type annotation is cumbersome for users, especially in large spreadsheets where manual annotation might become extensive. In contrast, their system requires fewer annotations that introduce the base units from which the re-

maining units can be inferred. They also argue that the system in [53] fails to detect some errors that their system does not.

Labels and dimensions can also be used in combination to locate errors in spreadsheets. Here, we mean labels similar to units as defined in UCheck [11] and dimensions as in units of measure e.g. meters squared or m^2. One could imagine a user trying to add different currencies or other units of measure. This is the approach taken by Chambers et al. [60, 66], and has been successfully implemented in F# [80]. They combine label and dimension inference [67] to check for errors that neither system would have discovered on their own. For example, adding the price per item of two different items is perfectly legal when considering only dimensions, but is illegal for labels as the addition is done using two different items such as apples and oranges. Conversely, adding quantities of apples is fine for labels, but if one quantity is specified in thousands, only dimension checking would catch the error. The paper does not discuss dimensions as ranges of cells, but as units of measurement such as those found in the SI unit system [81]. One benefit to using an existing system is that SI units can be combined and inferred based on pre-existing conventions, and so this system does not need additional annotation as was the case with Abramson et al.'s UCheck system and a system discussed later [53]. However, some annotation might still be needed in order to avoid ambiguous or dimensions that prove difficult to parse. On the other hand, units can be arbitrary rather than confined to the definitions of the SI system. The combination and textual representation of labels are taken from Erwig et al.'s work in [16] with some simplified rules. The system follows five steps: First, headers are inferred as described in [11, 77], and then labels are interpreted from the strings found in the headers. The system then assigns the horizontal and vertical axes to dimension and label checking. This is a common pattern in spreadsheets, where one table header denote some properties and the other axis denotes some objects that possess said properties. Dimension and label inference is then determined for each cell using a series of typing judgements and rules given in the paper. If no errors occur in this step, all dimensions are considered fully qualified. For example, the dimension $\{\texttt{m}, \delta\}$, where m signifies meters and δ signifies a missing dimension, can be both speed $\frac{m}{s}$ and acceleration $\frac{m}{s^2}$ based on the choice of δ. The most common choice is picked based on a heuristic.

Coblenz et al. [68] introduce their framework for error detection in spreadsheets called SLATE, or *A Spreadsheet Language for Accentuating Type Errors*, which closely resembles the work by Erwig et al. [16, 65]. In this paper, dimensions such as acceleration $\frac{m}{s^2}$ are referred to as *units*, while labels are properties such as an apple. To avoid confusion, we instead refer to these as *dimensions* and *units* respectively. As in [16], units can be generalised into a common ancestor type. For example, both apples and oranges are fruits. There are a few differences however. Users must manually enter dimension and unit information in parentheses in cells, requiring them to learn the system, and because the annotation is simply added parentheses, there is no highlighting or differentiation from actual cell content. Errors are propagated through operations, and a resulting label from a multiplication or division retains all labels of its operands. This means that a multiplication of apples and oranges results in the label: (apples, oranges) which is used to detect an error. For addition and subtraction, the label is instead the common ancestor of the labels of the operands: (fruit), since it is common in spreadsheets to add and subtract different objects.

Inspired by the theoretical foundations laid out by Erwig et al. in [16], Ahmad et al. [53] also developed a static type system. Their system differs from Erwig's because it requires explicit type annotations, and the former is also a type system based on units. The authors deviate from Erwig et al. by defining two relationships similar to those known from object-oriented programming: *is-a* and *has-a*. The first relationship describes units that are subcategories of some header. For example, a Toyota *is-a* car. The second describes unit properties, such as a car *has-a* steering wheel. Unit notation, rules and formal typing judgements are also introduced to reduce unit expressions to well-formed ones that the system can reason about. Operators for joining unit declarations on common relationships are also defined. Two advantages to explicit type annotation is that the system can accommodate uncommon spreadsheet layouts where it would be harder for the UCheck system to infer the

correct units and labels, and secondly the units are less prone to ambiguity since they are given. Although structured layouts are more common, this advantage should not be overlooked [11]. The generated error messages require the user to understand the *is-a* and *has-a* relationships to decipher the error messages and correct the mistakes. Constructing error messages that bridge the gap between formal type systems and end-users is a difficult problem, which could limit the system's commercial use. The authors note that an automated system is planned in future work.

In their experimental evaluation, the authors claim that they did not need to understand the problem domains to annotate the spreadsheets (23 spreadsheets in total from [82]). We argue that this might not always be the case in commercial settings, where domain knowledge is essential to annotate cells with sensible unit names. We found the set of spreadsheets used for the experiments relatively small. Nonetheless, the authors find a single, true error in the spreadsheets. The authors do however mention a larger, more detailed, empirical study as part of their future work. They also suggest a header inference algorithm that uses natural language processing and machine learning techniques that learn from user feedback.

Ahmad et al. [16] also compare their work with that of Erwig et al. [65]. They note that they describe a type system with weaker rules and that the lack of a distinction between the *is-a* and *has-a* relationships leads to incorrect unit inference. To support their claim, they present two examples where their system would have failed but Ahmad et al.'s system would not. On the other hand, Erwig et al. argue that their system catches errors that Ahmad et al.'s system does not, and that while the *has-a* relationship enables more fine-grained information, it complicates automatic header inference. Further work needs to be done in order to fully compare these works. Finally, we suggest a combination of automatic and manual type or unit annotation, to allow users to disambiguate cases where the automatic inference fails.

Lastly, we touch on an interesting observation. It would useful to have type systems that could be toggled, so that ordinary users are not overencumbered by information and that the system would normally only be used by expert users.

4.4 Testing

Testing is an essential and ubiquitous part of software development. It reassures us that the programs we construct do what they are supposed to do and/or fail on the appropriate input, although they cannot prove the absence of bugs. Given the large number of errors found in spreadsheets and the grave financial implications they have, there is a lot of research in this area. Testing calculations in a spreadsheet is important in order to ensure proper results in critical engineering and business applications. This re-emphasises the intuition that it is beneficial to apply software development principles to spreadsheets, as noted by Panko, and that spreadsheet development must embrace extensive testing in order to be taken seriously as a profession [49].

Testing is especially important for spreadsheets as several of the commercial and non-profit applications exhibit behaviour that would constitute an error in a software development context, but are nonetheless allowed. For example, summation across cell ranges silently ignore non-integer values (see figure 3 on page 11). Whether this should be treated as an error is up to the designers of the spreadsheet, but the reviewed literature favours treating these as errors [22].

A focus of much research is the "What You See Is What You Test" or WYSIWYT approach [21, 83], where the user incrementally tests the spreadsheet as it is being developed, and testing is complemented by visual feedback to guide the user through the process [14, 21, 69]. This enables errors to be caught earlier and rectified rather than later. The methodology is based on dataflow analysis and testing criteria of imperative programs. Rothermel et al. introduce the notion of *definition-use-adequacy* or *du-adequacy*. A *definition* is the location of a cell's value and a *use* is the location of the usage of that cell. Together they form a *du-pair*, and a cell can have multiple *du-pairs* associated with it if there is more than one cell using its definition. The *du-adequacy* of a cell is thus the degree of exercised *du-pairs* that directly or transitively reference a cell's definition and so contributes to its output value. This is extrapolated to include all *du-pairs* such that each pair is exercised and they influence a given output cell. *Du-associations* and execution traces are com-

bined in order to determine the degree to which a cell is tested. The *du-associations* are given by the *cell relation graph*, which resembles the support graph defined by Sestoft in [2], but whose edges are only defined by the relation between formulas. To invoke the system, a user clicks on a validation tab in a cell to tell the system that the cell's value is correct. This action is propagated through the cells that contribute to this cell's output. The validation tab contains a question mark if a cell's output was previously tested, but a change in the spreadsheet requires that it be retested. If the cell is fully tested, it contains a checkmark instead. The degree to which this adequacy criterion is satisfied determines the border colours of cells in the spreadsheet. Cells that are more tested appear bluer and less tested cells appear redder. The border colours of these cells are therefore similar to the degree of test coverage, a term found in software development where it denotes the percentage of code paths that have been tested in a project. For example, cells using non-strict functions such as `IF` will only be fully tested if both branches have been taken. The authors' choice of colours can be drawn into question. In our opinion, the combination of red and green provide a better notion of contrast and meaning for end-users as these colours are commonly found in our everyday lives (e.g. traffic lights and entrance-exit signs.). However, this is also a matter of taste and aesthetics, so there is no inherently "correct" choice. Another issue is cells that have a range of correct values where a single test may not be representative of its correctness: The test might succeed for a couple of values within the range, but fail for others. Others have investigated automatic test case generation for spreadsheets which could be combined with the WYSIWYT methodology [70, 71, 84].

Property-based testing has seen much interest since the invention of the Haskell library QuickCheck [85] for automatically generating input values for tests, alleviating the burden of devising test cases. Property-based testing would be attractive in a spreadsheet context as well, especially when considering user-defined functions, which we explore in section 5.

Fisher et al. [71] have developed a system for automatic test case generation based on the WYSIWYT methodology, and implemented in the Forms/3 spreadsheet language [20]. The system only handles integer values. Untested cells are initially marked by a red border. The user selects one or more cells to test, and the cell is tested with the current input and marked as tested by a check mark. In addition, the colour of its border, and that of all other cells that transitively helped produce the final result, is changed to a shade of blue to reflect their test coverage, where opaque blue represents full coverage. This is very similar to the work in [21, 83]. The system keeps track of test coverage via cell references. The system can optionally enhance the visual feedback by displaying dataflow arrows, which is found in most spreadsheet applications, using the colouring scheme just described. Any modification to a tested cell will emit a change of the colour if some cells subsequently need to be retested. The end-user thus has a highly intuitive, visually enhanced, testing framework available which can be used incrementally during spreadsheet development. The test case generation process is initiated by pressing a "Help Me Test" button, and the user can refine the test case generation by selecting a subset of cells that he or she wishes to test. The system then attempts to generate test cases that exercise the *du-associations* that are involved in the selected output cells. Although not explained in detail in the paper, it is assumed that the user will then validate the outputs resulting from the generated tests with the WYSIWYT methodology.

The authors present two approaches to test case generation. A straight-forward *random* approach, where sample input values to cells are generated randomly, and a more intelligent, *goal-oriented* approach that uses constraints and branch functions to generate more meaningful test cases. For random test case generation it may be difficult to generate appropriate input values that ensure that user-selected cells are tested properly, but on the other hand it is simpler to implement and may provide satisfactory results for most trivial scenarios. Conversely, the goal-oriented approach is more complex, but may provide better test cases. The goal-oriented method is a simplified version of the *Chaining approach* by Ferguson et al. [70]. Both approaches were extended with range information which the authors included themselves based on inspection of the formulas in the spreadsheets. Ultimately, the goal-oriented approach proves most effective according to a series of empiri-

cal studies without any range information for test input data providing the best percentage of test case generation coverage with a 100% coverage on feasible *du-associations* on half of the spreadsheets (10 in total), and the same method with range information having the better performance. In general, the two *Chaining* approaches outperformed the random test case generation strategy. We make one crucial observation: Since property-based testing relies heavily on a solid type system to infer appropriate test cases, it would be sensible to assume that a robust type system for spreadsheets would benefit such a testing framework.

The papers discussed thus far have been concerned with *testing* spreadsheets. In contrast, Abraham et al. lay the theoretical groundwork for a system for *debugging* spreadsheets called GoalDebug [86] and substantiate it in [69]. The framework lets users input the expected value for a cell that outputs an incorrect value and are then given a list of suggested changes that will yield the expected value in that cell. The list is ranked using heuristics to provide the user with the best solutions first.

The system provides a graphical interface, from which the GoalDebug system can be initialised. Different types of change suggestions can be achieved using different strategies defined by the authors, and different heuristics are used for different suggestions. Examples of a change suggestion would be changing a reference to another or replacing a constant. Constraints are simplified as much as possible to simplify constraint solving.

The authors define *copy-equivalence* where two formulas contain the same relative references and one could therefore have been created from the other by copy. This is analogous to the *cp-similarity* defined by Abramson and Erwig in [14] and *clones* defined by Hermans et al. [56]. Copy-equivalence is considered more precise than structural equivalence and is used to rank suggestions. References that are closer by Manhattan distance are also ranked higher by the assumption that proximity and relevance are correlated.

The GoalDebug system is interesting because it deals with guiding the user through solving his or her mistakes (as one would expect from a software debugger) while the other literature in this section informs the user that something is wrong, but not necessarily what the cause is or how the problems should be rectified. For example, a message: "Type conflicts: In D8: expected **Num**, found **Undef**" from [14] does not tell the user how to fix the conflict nor does it tell him or her where the problem originates in the spreadsheet. This observation is not meant to devalue the research in the other papers, but simply highlights an important difference from the viewpoint of end-users. The authors integrate the UCheck system from [11] to rank suggestions that keep units intact higher than those who do not. Presumably using the UCheck system, all change suggestions are type-checked in order to ensure that they do not introduce type conflicts if they are introduced.

The paper reports that GoalDebug is effective at generating suggestions that correctly recover from spreadsheet errors and at ranking change suggestions. The authors intend to conduct empirical studies with end-users to evaluate the system's usability, and also suggest combining WYSIWYT with GoalDebug.

4.5 Assertions

Assertions are usually used in tests or to ensure that the program never enters an illegal state. They test some condition and if that condition proves false, the program is terminated. For example, `assert (x == 1)` will terminate the program if `x` is not equal to one. In this section, we will look at how assertions can be used in spreadsheets.

To start things off, we examine a simplistic approach used in the Funcalc project. A particular worksheet is used for testing, where certain cells contain a 1 if the output matched the expected output, 0 otherwise, similar to assertions but without terminating the program. The spreadsheet then needs to be manually inspected in order to assess the result of the tests. In software development, it is commonplace for a test tool to automatically collect all tests in a project, execute them and then output a report that recapitulates the results as well as a multitude of other useful features. The lack of a standardised testing tools may be one reason as to why software developers frown upon spreadsheets. A slightly more sophisticated approach is taken by Gnumeric [6], where a collection of Perl scripts check one or more cell values after a spreadsheet

has been evaluated [87]. In this sense, testing is slightly more automated but test spreadsheets still need to be created and tests are created in an external language with a steep learning curve for end-users. The simplistic approaches taken by Funcalc and Gnumeric are effective from an end-user usability perspective since they can inspect and verify values themselves without any training.

Burnett et al. [51] implement assertions for spreadsheets in the Forms/3 spreadsheet language [20]. Contrary to the two previous approaches, these assertions are an integrated part of the spreadsheet and are specifically geared towards end-user development. Assertions on cell values can be defined by the user, and these assertions are automatically propagated through the implicit dataflow graph in the spreadsheet using logical reasoning and interval arithmetic (the authors do not explain this process in detail). Consequently, the system can also report *assertion conflicts* due to propagated assertions that do not agree on an output, apart from a cell's assertions that report an erroneous value in that cell. The propagation takes cell formulas into account: If cell A has the assertion that its value must be in the interval $[1, 10]$ and cell B contains the formula $A - 1$, then the propagated assertion on cell B is $[0, 9]$. This is very useful as assertions can be defined on input cells that are part of a long set of dependent cells that use these inputs for intermediate computation. Otherwise, the user would need to define all these assertions manually. In software terminology, a cell's own assertion is its *postcondition* that must hold after the cell's contents has been evaluated, while the upstream assertions that affect the cell are its *preconditions* that must hold before the cell is evaluated. Empirical studies have shown that users were comfortable with using assertions and that they discouraged overconfidence in users, something that behavioural science denotes as common in software development and end-user programming, and in humans in general.

5 Functional Spreadsheets

5.1 User-defined Functions

Most modern spreadsheet applications already allow users to define their own functions. For example, Excel permits user-defined functions through Visual Basic and can interface with other external languages, and attempts have even been made to embed programs in cells [18]. Visual Basic can also be slow [15]. However, writing and debugging functions in an external language requires end-users to understand a programming language and thus requires a high investment. Many of the implementations in the literature use an external language to interface with the spreadsheet application. Cheng et al. used VBA as glue between an OCaml implementation of their type system and Excel [23]. Of course, IT professionals can write the required functions for the end-users at the cost of efficiency and control [2]. Research has thus attempted to find alternative ways of letting end-user programmers define their own functions, but what are some of the benefits to user-defined functions? First, they embody the *don't-repeat-yourself* or DRY principle from software development that discourages unnecessary repetition of code, which in turn lowers the risk of errors. Second, they create a logical and physical location for function code and its accompanying documentation that is easy to distribute. Third, functions can be compiled to high-performance byte- or machine code, and in fact this is what happens in Funcalc [19].

5.2 Sheet-defined Functions

Research has investigated user-defined functions that can be defined directly in the spreadsheet [2, 15]. Peyton-Jones describes these as user-defined functions and argues that functions must be defined in the spreadsheet "*because it is the only computational paradigm understood by our target audience*" [15], but this is not a suitable term as it uses the same name for referring to functions defined in an external language. Sestoft coined the term *sheet-defined* functions for this exact purpose, so to avoid confusion, we henceforth refer to functions defined directly in the spreadsheet as *sheet-defined* functions and those that are defined in an external language as *user-*

defined functions.

Sheet-defined functions are a relatively new concept, first introduced in the Forms/3 spreadsheet language [15, 20]. There are additional benefits to sheet-defined functions [2, 15]: Functions are defined directly in the spreadsheet using the concepts that end-users are already familiar with, and so require a lower investment than using user-defined functions, and bestows some self-documenting qualities to the functions. Additionally, documentation can be localised to the function definition. It is also more straightforward to add sheet-defined functions to older spreadsheets that lack them, and can be done by end-users. Sharing functions is a matter of sharing the function definitions in worksheets and not binary files such as shared libraries (although this would certainly be possible). Sheet-defined functions also do not have any side effects like some external languages, because they must respect the restriction that you cannot modify a cell from another cell. Immutability has long be heralded by functional programming advocates as promoting correctness, among other things [26]. The function bodies can be separated on different lines to make it easier to understand the purpose of the function, and avoids single-cell monolithic formulas. The programming language equivalent would be a function with intermediate calculations on separate lines versus a one-liner. Finally, they do not break the audit trail, i.e. the list of recorded changes to a cell.

To further emphasise the advantages of sheet-defined functions, consider a programming language where you cannot define your own functions. This reflects the lack of functions in end-user development in spreadsheets. As eloquently put by Peyton-Jones et al.:

> "*Can you imagine programming in C without procedures, however clever the editor's copy-and-paste technology?*" [15]

One of Casimir's criticisms [17] of the Lotus 1–2–3 spreadsheet software was its lack of user-defined functions which he argued would increase memory consumption because copies of formulas must be stored in each cell. A single copy stored in a sheet-defined function would be much more sensible and less error-prone as we have already mentioned. Sheet-defined functions also complicate matters. How should higher-order functions be represented in a primarily first-order language? Should recursive functions be allowed? Are these concepts too difficult for end-users to understand? In the following discussion of the literature we will attempt to answer some of these questions.

Peyton-Jones et al. [15] proposed a design of sheet-defined functions. Functions are defined in special *function instance sheets* that represent a single instance of the function. As the name implies, function instance sheets are instances of the function invocations. This means that each invocation has a copy of the function definition; there is no single definition of a function. The authors argue that this lowers the learning cost as function instances behave like ordinary worksheets. Consequently, when a user edits a function, a pop-up appears that prompts the user for editing this single function instance or all function instances. This is reminiscent of some calender systems and repeated events, where you can choose to modify a single instance of the event or all of its repetitions. They can also be defined using a graphical interface and defined based on existing formula in the spreadsheet. When editing the function, graphical user interface tools are provided to easily navigate up and down the call tree. This approach raises some interesting questions. If a user changes a single function instance but not its name, there are now two or more functions where one single function behaves differently from the rest. However, the system then changes all the invocations of the function. While this is fine, it breaks the separation of the function instances, yet the alternative (having multiple functions with the same name but different behaviour) would increase memory consumption and unnecessarily complicate future edits because the system now has to distinguish different functions with the same name. They incorporate program usability studies from human-computer interaction research, namely the *Cognitive Dimensions of Notations* [88, 89] and *Attention Investment* models [90], to assess multiple criteria of their proposed design of sheet-defined functions and to determine the likelihood of users adopting their approach. It is a known problem that end-users do not wish to expend too much time and energy when trying to learn a new system, so this criterion is crucial for the success of sheet-defined functions.

Letting end-users define functions with the tools they already know, lowers their entry-barrier.

Inspired by the work of Peyton-Jones et al., Sestoft [91] implemented sheet-defined functions in Funcalc [19]. One significant inspiration was the performance aspect mentioned by Peyton-Jones et al.: That functions could be compiled to byte code or executed by a just-in-time compiler. While Peyton-Jones et al. implement their design in VBA, sheet-defined functions in Funcalc are compiled to efficient `.NET` bytecode, more specifically the *Common Intermediate Language* or CIL, by an internal compiler using the runtime code generation facilities of C#. VBA is both interpreted and compiled to Microsoft p-code (pseudo-code) [92], and but initial experiments indicate that executing .NET bytecode is comparable to or faster than VBA[91]. In contrast to the design of Peyton-Jones et al., functions have a single definition and are cached and reused. Functions are defined in *function sheets*, worksheets whose names are prefixed with an @ sign and whose tabs are pink. Since each function instance sheet in Peyton-Jones et al. is separate, function parameters use inter-worksheet references. In Sestoft's function sheets, a call to the `DEFINE` function defines a new function, e.g. `=DEFINE("F", out,` $input_1$`, ...,` $input_n$`)` defines the function `F` with an output cell and zero or more input cells. An example is depicted in figure 4. Performance results show competitive and even improved runtime performance over functions in Microsoft Excel [2].

34	=DEFINE("ndie", B36, B35)	'General n-side die
35	'n =	6
36	'eyes =	=FLOOR(RAND()*B35, 1)+1

Figure 4: A example of a sheet-defined function in Funcalc that calculates a random value using an n-sided die [2]. The call to `DEFINE` is on the first line along with a short description, and the in- and output parameters are defined on the next couple of lines.

In a video presentation at the 2009 *Commercial Users of Functional Programming* or CUFP conference, Lee Benfield [93] presents the *Functional Model Deployment* or FMD framework for Excel that attempts to retain the expressiveness of spreadsheets while provided users with auto-generated code for communicating with external libraries without resorting to VBA, or any other external language. It allows variable declarations and function handles. For example, `variable("date")` creates a single variable called "date", and `@@11` refers to a function handle. Once available, functions can be evaluated with arguments using an `eval` function. Like Sestoft's sheet-defined functions, FMD introduces functions from functional programming such as `map` as an alternative to copy-pasting cell formulas, which we have already discussed is a source of errors in section 4. Like Peyton-Jones and Sestoft, FMD also introduces higher-order functions, but unlike them, FMD provides tuples, an important concept in functional programming. When Excel traverses the underlying graph of cells, FMD generates the boilerplate code that communicates with other libraries, thus removing the need for VBA glue code that would normally be needed to use these libraries in Excel.

The major conceptual difference in Benfield's work, lies in his more direct approach to implementing functional programming in spreadsheets. This means that the power and expressiveness of functional programming can be leveraged by users and variables encourage additional reuse alongside functions. The downside is the investment people have to make in order to learn concepts such as `map`. Hoon takes a similar approach [22] which we discuss later. Peyton-Jones and Sestoft take a more user-oriented approach where they attempt to reuse the already familiar spreadsheet concepts to implement higher-order functions.

Sheet-defined functions introduce new challenges. We mentioned earlier that spreadsheets are first-order functional languages, but functional languages support higher-order functions that greatly increase expressiveness. Should higher-order functions be allowed in spreadsheets? Will their introduction cause more confusion than good? Their advantages should be clear. For example, a sheet-defined function can be passed to functions such as `COUNTIF` and `SUMIF` to obtain new, more advanced functionality, particularly when combined with partial function evaluation. Modifying such a function would involve a single edit in its corresponding function sheet, while modifying the formula would need to be done for all instances. Peyton-Jones et al. do not discuss higher-order functions.

Sestoft implements this using the `CLOSURE` and `APPLY` functions. A closure for the `NDIE` function in figure 4 is defined by `CLOSURE("NDIE", 20)` and returns a function value, which can then be called using `APPLY(ca)` where `ca` is the cell address containing the closure. Alternatively, the closure can be given directly in a call to a function such as `COUNTIF`. It is also possible to define partially applied functions.

Hoon et al. implemented a spreadsheet application in the functional, lazy and higher-order language Clean [94] in order to evaluate a spreadsheet with such functional properties [22], as well as a symbolic evaluator for equations which would be useful for the financial and scientific communities. The resulting application was called FunSheet. While functional concepts are certainly powerful, there are things such as `map` and variations of `fold` that will demand a high investment cost from end-users. Since Clean already uses term graph rewriting systems under the hood, cyclic references are easy to express, and so all function applications are thus replaced with their definition or with a predefined function as necessary.

Another aspect is recursion. Is it possible for these types of functions to support recursion, and should they? Casimir claims that recursion is problematic because it may require many recalculations to iteratively compute a result [17]. Yoder et al. [18] dispute this claim, and argue that natural-order recalculation, i.e. calculating all dependencies of a cell before evaluating the cell itself akin to topological sorting, elegantly solves this problem. They do note though that the lack of formula-local variables can lead to considerable memory usage as intermediate values can only be stored in the "global" memory of the spreadsheet cells. Let us assume for now that recursion can be implemented efficiently in the spreadsheet paradigm, then the question remains if it should. Recall that a substantial amount of the discussed literature attempts to use principles from the software development and computer science fields in a manner that is transparent or at least easier for the user to understand since they lack a formal education in IT. An example is using units instead of types to detect errors [16, 65]. Recursion is a useful tool for solving problems that can be defined recursively but requires that end-users can grasp the concept in order to use it. Regardless, it would undoubtedly be invaluable for more advanced users or those with an IT background. Peyton-Jones et al. also argue that the lack of inductive types (besides the integer) means that recursion might not be as expressive as it is in functional languages [15]. In the case of Hoon et al.'s work, their spreadsheet is already defined using a functional language and recursion is readily available. They deal with recursion similarly to how the Y-combinator is used in lambda calculus. Sestoft's sheet-defined functions are fully capable of recursion and are also tail-recursive, but there is currently no guard against infinite recursion although this is a purely practical problem.

As described in section 4.3.3, Burnett et al. [65] suggested a *unit view* for spreadsheets, similar to how one can view formulas in a formula view. This would be very useful in general, but also in combination with sheet-defined functions as one can more accurately determine the source of type errors. In Funcalc [19], a type error in a sheet-defined function is signalled by the error value `#ERR: ArgType`. The user has no information about the position or expected type of the erroneous argument. A unit view that is aware of sheet-defined functions would help in this case.

6 Visualisation

Visualisation techniques are important due to the invaluable and intuitive feedback that they can provide to end-users. Visual feedback is important for most, if not all, aspects of spreadsheets since a big part of spreadsheets is their visual representation of data. It provides visible feedback for errors and dataflow through cells and worksheets, and their very nature makes these types of tools easy to grasp.

We do not discuss visualisation in terms of the graphical user interface or its enhancements in spreadsheet applications, nor graphical reporting facilities such as graphs or charts. We only discuss visualisation in terms of tools that are specific to spreadsheets and provide some benefit to spreadsheet end-users in terms of the categories we have previously discussed.

Dataflow diagrams have been used to give a high-level view of the inter-worksheet relations of a spreadsheet, to allow end-users to reason what the structure of a spreadsheet and more clearly explain their intentions. Hermans et al.

[95] developed such a tool called Breviz. It is common that end-users inherit spreadsheets from co-workers within an organisation and must spend some time deciphering the spreadsheet. Dataflow diagrams are a means to alleviate this process. The dataflow diagrams are annotated with arrows to depict the inter-worksheet references. The thickness of the arrows are proportional to the number of cell references between worksheets. This is a style related to the dataflow that is used for program analysis. Hermans et al. completed a series of empirical studies at the Dutch asset management company Robeco. They found that almost all of their interviewees understood the dataflow diagrams and there was a 80% consensus that dataflow diagrams would be beneficial in their daily work. Some participants noted that the diagrams lacked information such as the filtering of data from one worksheet to another, while others felt that they needed more time to study the diagrams to assess their merit. One participant wanted the visualisation to be within Excel to get a better overview. The authors also found that when a spreadsheet was transferred between people, Breviz helped give the recipient a high-level layout of the spreadsheet and its intended purpose. The same was true for a case study with spreadsheet auditors. In later work, Hermans et al. enhance the dataflow diagrams in Breviz with inter-worksheet code smells [56], as described in section 4.1.

We briefly mention a commercial counterpart to Breviz, SLATE [96], not to be confused with the testing framework we discussed in section 4.4 with the same name [68]. It is an Excel plugin and provides added features to Breviz. SLATE opens a new window with its visualisation. SLATE shows you formula dependencies and highlights its relevant parts when the mouse is hovered above the formula's subsections. For example, when hovering over the condition in an `IF` function, SLATE will highlight the cell or cells that are used to determine the condition. Pressing a cell address in SLATE will take you back to Excel and momentarily highlight that cell.

7 Related Work

To the author's knowledge, no comprehensive literature review of spreadsheet technology exists. Sestoft's book on spreadsheet technology [2] describes many different aspects, but its main focus is on the implementation details of spreadsheet technology, notably Funcalc, and is not intended to serve as a general literature review of spreadsheet technology. Biermann [97] surveys approaches to declarative parallel programming in spreadsheets using array programming, but does not cover general spreadsheet technology. It is the intention of this paper to provide such a review with a satisfactory coverage of existing spreadsheet technology, and bring readers up to speed on spreadsheet technology.

8 Conclusion

In this study, we have examined the current challenges in spreadsheets and the research that has tried to overcome them. The main challenges have been weak type systems, unruly error reporting, and a lack of standardised tools for testing and debugging, best practice guidelines and strong type systems. The cumulative effect of these shortcomings is likely the reason why software developers do not seem to hold spreadsheets in high regard, and even regard spreadsheets as boring [17]. We believe that the surveyed research not only makes spreadsheets more sophisticated and usable, but could help change how software developers view spreadsheets because it draws on principles of software development and computer science that they are familiar with. Burnett and others have coined this as *end-user software engineering* [98, 99] where end-user programming and development incorporate the principles from traditional software engineering which ensure the same level of reliability, efficiency and usability [100] as found in software development. This is usually done in a manner that is user-friendly or even fully transparent to the user, so that they can take full advantage of the system, while lowering the investment required to learn it. One notable strategy for achieving this is the use of (immediate) visual feedback such as with the WYSIWYT methodology [21, 83] or dataflow diagrams [10, 78, 95]. Furthermore, Burnett et al. remark that their approach has been "*. . . to gently alert them [end-users] to dependability problems, to assist them with their explorations into those problems to whatever extent they choose to pursue such ex-*

plorations, and to work within the contexts with which they are familiar" [98]. This approach has been adopted by the majority of the covered literature and is a combination of the best of both worlds.

Another notable example of end-user software engineering is *sheet-defined* functions that promise to give end-users a higher degree of expressiveness, e.g. through the use of higher-order functions, and modularity and reuse of functions defined in a paradigm that they are already acquainted with, steering clear of the often inconvenient need for collaboration with an IT department.

We conclude with a summarising list of high-level observations.

- Efficient data structures are vital for efficient recalculation and spatial representation of cells (section 2).
- Lazy evaluation of cell expressions is an attractive prospect in terms of performance and visualisation, but more work needs to be done in order to evaluate this strategy and identify challenges (section 3).
- Parallel recalculation promises to speed up the recalculation process, but has seen less interest than expected in relation to other subjects (section 3.2).
- Templates ensure spreadsheets that are free from errors and provide a common basis for a set of similar spreadsheets (section 3.3).
- Bugs in spreadsheets are abundant (section 4)
- Bug detection and elimination is by far the most prominent subject for spreadsheets in terms of research activity, due to the previous point (section 4 and the following sections).
- Error and smell classification is not standardised (section 4.1).
- Most type systems in spreadsheet applications are weak and afford few guarantees about the correctness of expressions (section 4.3).
- There seems to be a balance to be struck between static and dynamic (i.e. inferred) type systems (section 4.3).
- A "unit view" for types, similar to a formula view, is an interesting idea that should be investigated further, especially in combination with sheet-defined functions section 4.3.3.
- An interesting direction for future work would be to explore how different type systems and error detection systems can be integrated and work in harmony.
- A majority of the surveyed literature uses software development and computer science techniques as a foundation for their work.
- Sheet-defined functions show promise in bridging the gap between IT professionals and end-users, and allowing the latter group to create powerful functions. They also bring about new challenges (section 5.2).

9 Acknowledgements

The author would like to thank Peter Sestoft for his guidance and helpful suggestions during the writing of this literature review.

References

[1] Christopher Scaffidi, Mary Shaw **and** Brad Myers. "*Estimating the numbers of end users and end user programmers*". In: *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*. IEEE. 2005, pp. 207–214.

[2] Peter Sestoft. "*Spreadsheet Implementation Technology*". The MIT Press, 2014. ISBN: 9780262526647.

[3] Tomás Isakowitz, Shimon Schocken **and** Henry C. Lucas Jr. "*Toward a Logical/Physical Theory of Spreadsheet Modeling*". In: *ACM Trans. Inf. Syst.* 13.1 (Jan. 1995), pp. 1–37. ISSN: 1046-8188. DOI: 10.1145/195705.195708. URL: http://doi.acm.org/10.1145/195705.195708.

[4] D. J. Power. "*A Brief History of Spreadsheets*". URL: http://www.dssresources.com/history/sshistory.html (visited on 06/21/2016).

[5] The Document Foundation. "*LibreOffice Calc*". URL: https://www.libreoffice.org/discover/calc/ (visited on 05/09/2016).

[6] The GNOME Project. "*Gnumeric*". URL: http://www.gnumeric.org/ (visited on 06/02/2016).

[7] Apple Inc. "*Apple Numbers*". URL: http://www.apple.com/dk/mac/numbers/ (visited on 05/09/2016).

[8] Dan Bricklin **and** Bob Frankston. "*VisiCalc: Information from its creators, Dan Bricklin and Bob Frankston*". URL: http://www.bricklin.com/visicalc.htm (visited on 06/16/2016).

[9] Google Inc. "*Google Spreadsheets*". URL: https://www.google.com/sheets/about/ (visited on 06/02/2016).

[10] F. Hermans, M. Pinzger **and** A. **van** Deursen. "*Detecting and visualizing inter-worksheet smells in spreadsheets*". In: *2012 34th International Conference on Software Engineering (ICSE)*. June 2012, pp. 441–451. DOI: 10.1109/ICSE.2012.6227171.

[11] Robin Abraham **and** Martin Erwig. "*UCheck: A spreadsheet type checker for end users*". In: *Journal of Visual Languages & Computing* 18.1 (2007), pp. 71–95. ISSN: 1045-926X. DOI: http://dx.doi.org/10.1016/j.jvlc.2006.06.001. URL: http://www.sciencedirect.com/science/article/pii/S1045926X06000383.

[12] Brian Harvey **and** Matthew Wright. "*Simply Scheme: introducing computer science*". Mit Press, 1999.

[13] Brian Harvey **and** Matthew Wright. "*Simply Scheme: Introducing Computer Science Website*". URL: https://people.eecs.berkeley.edu/~bh/ss-toc2.html (visited on 06/15/2016).

[14] Robin Abraham **and** Martin Erwig. "*Type Inference for Spreadsheets*". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '06. New York, NY, USA: ACM, 2006, pp. 73–84. ISBN: 1-59593-388-3. DOI: 10.1145/1140335.1140346. URL: http://doi.acm.org/10.1145/1140335.1140346.

[15] Simon Peyton-Jones, Alan Blackwell **and** Margaret Burnett. "*A User-centred Approach to Functions in Excel*". In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. New York, NY, USA: ACM, 2003, pp. 165–176. ISBN: 1-58113-756-7. DOI: 10.1145/944705.944721. URL: http://doi.acm.org/10.1145/944705.944721.

[16] Martin Erwig **and** Margaret Burnett. "*Adding apples and oranges*". In: *Practical Aspects of Declarative Languages*. Springer, 2002, pp. 173–191.

[17] Rommert J. Casimir. "*Real Programmers Don't Use Spreadsheets*". In: *SIGPLAN Not.* 27.6 (June 1992), pp. 10–16. ISSN: 0362-1340. DOI: 10.1145/130981.130982. URL: http://doi.acm.org/10.1145/130981.130982.

[18] A. G. Yoder **and** D. L. Cohn. "*Real spreadsheets for real programmers*". In: *Computer Languages, 1994., Proceedings of the 1994 International Conference on*. May 1994, pp. 20–30. DOI: 10.1109/ICCL.1994.288396.

[19] Peter Sestoft. "*Corecalc and Funcalc Spreadsheet Technology in C#*". In: (2014). URL: http://www.itu.dk/people/sestoft/funcalc/ (visited on 06/01/2016).

[20] Allen Ambler. "*Forms: Expanding the visualness of sheet languages*". In: *1987 Workshop on Visual Languages*. 1987, pp. 105–117.

[21] Gregg Rothermel **et al.** "*What You See is What You Test: A Methodology for Testing Form-based Visual Programs*". In: *Proceedings of the 20th International Conference on Software Engineer-*

ing. ICSE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 198–207. ISBN: 0-8186-8368-6. URL: http://dl.acm.org/citation.cfm?id=302163.302183.

[22] Walter A.C.A.J. **de** Hoon, Luc M.W.J. Rutten **and** Marko C.J.D **van** Eekelen. "*Implementing a Functional Spreadsheet in Clean*". In: *Journal of Functional Programming* 5 (1995), pp. 383–414.

[23] Tie Cheng **and** Xavier Rival. "*Static Analysis of Spreadsheet Applications for Type-Unsafe Operations Detection*". In: *European Symposium On Programming (ESOP'15)*.

[24] Morten Poulsen **and** Poul Peter Serek. "*Optimized Recalculation For Spreadsheets With the Use of Support Graph*". Master Thesis. IT University of Copenhagen, 2007.

[25] Hildur Uffe Flemberg **and** Martin Jeanty Larsen. "*Optimizing Spreadsheet Computations on Multi-Core Architectures*". Master Thesis. IT University of Copenhagen, Sept. 1, 2014.

[26] John Backus. "*Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs*". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. DOI: 10.1145/359576.359579. URL: http://doi.acm.org/10.1145/359576.359579.

[27] Arvind **and** R.S. Nikhil. "*Executing a program on the MIT tagged-token dataflow architecture*". In: *Computers, IEEE Transactions on* 39.3 (Mar. 1990), pp. 300–318. ISSN: 0018-9340. DOI: 10.1109/12.48862.

[28] C.H. **van** Berkel, M.B. Josephs **and** S.M. Nowick. "*Applications of asynchronous circuits*". In: *Proceedings of the IEEE* 87.2 (Feb. 1999), pp. 223–233. ISSN: 0018-9219. DOI: 10.1109/5.740016.

[29] Vivek Sarkar. "*Partitioning and Scheduling Parallel Programs for Multiprocessors*". Research Monographs In Parallel and Distributed Computing. Cambridge, Massachusetts: MIT Press, 1989. ISBN: 0262691302.

[30] Vivek Sarkar **and** John Hennessy. "*Compile-time Partitioning and Scheduling of Parallel Programs*". In: *SIGPLAN Not.* 21.7 (July 1986), pp. 17–26. ISSN: 0362-1340. DOI: 10.1145/13310.13313. URL: http://doi.acm.org/10.1145/13310.13313.

[31] Vivek Sarkar **and** John Hennessy. "*Compile-time Partitioning and Scheduling of Parallel Programs*". In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. SIGPLAN '86. New York, NY, USA: ACM, 1986, pp. 17–26. ISBN: 0-89791-197-0. DOI: 10.1145/12276.13313. URL: http://doi.acm.org/10.1145/12276.13313.

[32] David Cann. "*Retire Fortran? A debate rekindled*". In: *Communications of the ACM* 35.8 (Aug. 1992), pp. 81–89.

[33] Vivek Sarkar **and** David Cann. "*POSC-a Partitioning and Optimizing SISAL Compiler*". In: *SIGARCH Comput. Archit. News* 18.3b (June 1990), pp. 148–164. ISSN: 0163-5964. DOI: 10.1145/255129.255152. URL: http://doi.acm.org/10.1145/255129.255152.

[34] Gilles Kahn. "*The semantics of a simple language for parallel programming*". In: *In Information Processing '74: Proceedings of the IFIP Congress*. Vol. 74. 1974, pp. 471–475.

[35] N. Halbwachs **et al.** "*The synchronous data flow programming language LUSTRE*". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320. ISSN: 0018-9219. DOI: 10.1109/5.97300.

[36] Vivek Sarkar **and** John Hennessy. "*Compile-time Partitioning and Scheduling of Parallel Programs*". In: *SIGPLAN Not.* 21.7 (July 1986), pp. 17–26. ISSN: 0362-1340. DOI: 10.1145/13310.13313. URL: http://doi.acm.org/10.1145/13310.13313.

[37] Microsoft. "*Task Parallel Library*". URL: https://msdn.microsoft.com/da-dk/library/dd460717.aspx (visited on 08/12/2016).

[38] David Abramson **et al.** "*ActiveSheets: Super-computing with spreadsheets*". In: *2001 High Performance Computing Symposium (HPC '01), Advanced Simulation Technologies Conference*. Citeseer. 2001, pp. 22–26.

[39] Krishna Nadiminti **et al.** "*ExcelGrid: A .NET plug-in for outsourcing Excel spreadsheet workload to enterprise and global grids*". In: *Proceedings of the 12th International Conference on Advanced Computing and Communication (ADCOM 2004)*. 2004.

[40] D. Abramson **et al.** "*Nimrod: a tool for performing parametrised simulations using distributed workstations*". In: *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*. Aug. 1995, pp. 112–121. DOI: 10.1109/HPDC.1995.518701.

[41] Axcelon Inc. "*EnFuzion*". URL: http://www.axceleon.com/prod-enfuzion/ (visited on 07/27/2016).

[42] SpreadsheetGear LLC. "*SpreadsheetGear*". URL: http://www.spreadsheetgear.com/company/about.aspx (visited on 06/30/2016).

[43] Microsoft. "*HPC Services For Excel*". URL: https://technet.microsoft.com/en-us/library/ff877820(v=ws.10).aspx (visited on 06/30/2016).

[44] M. Erwig **et al.** "*Automatic generation and maintenance of correct spreadsheets*". In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* May 2005, pp. 136–145. DOI: 10.1109/ICSE.2005.1553556.

[45] Martin Erwig **et al.** "*Gencel: A Program Generator for Correct Spreadsheets*". In: *J. Funct. Program.* 16.3 (May 2006), pp. 293–325. ISSN: 0956-7968. DOI: 10.1017/S0956796805005794. URL: http://dx.doi.org/10.1017/S0956796805005794.

[46] Robin Abraham **and** Martin Erwig. "*Inferring Templates from Spreadsheets*". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. New York, NY, USA: ACM, 2006, pp. 182–191. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134312. URL: http://doi.acm.org/10.1145/1134285.1134312.

[47] Robin Abraham **et al.** "*Visual specifications of correct spreadsheets*". In: *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*. IEEE. 2005, pp. 189–196.

[48] Raymond R. Panko. "*What we know about spreadsheet errors*". In: *Journal of Organizational and End User Computing (JOEUC)* 10.2 (1998), pp. 15–21.

[49] Raymond R. Panko. "*What we dont know about spreadsheet errors today*". 2015.

[50] Stephen G. Powell, Kenneth R. Baker **and** Barry Lawson. "*A Critical Review of the Literature on Spreadsheet Errors*". In: *Decis. Support Syst.* 46.1 (Dec. 2008), pp. 128–138. ISSN: 0167-9236. DOI: 10.1016/j.dss.2008.06.001. URL: http://dx.doi.org/10.1016/j.dss.2008.06.001.

[51] M. Burnett **et al.** "*End-user software engineering with assertions in the spreadsheet paradigm*". In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. May 2003, pp. 93–103. DOI: 10.1109/ICSE.2003.1201191.

[52] James Reichwein, Gregg Rothermel **and** Margaret Burnett. "*Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging*". In: *SIGPLAN Not.* 35.1 (Dec. 1999), pp. 25–38. ISSN: 0362-1340. DOI: 10.1145/331963.331968. URL: http://doi.acm.org/10.1145/331963.331968.

[53] Y. Ahmad **et al.** "*A type system for statically detecting spreadsheet errors*". In: *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. Oct. 2003, pp. 174–183. DOI: 10.1109/ASE.2003.1240305.

[54] "*Report of JPMorgan Chase & Co. management task force regarding 2012 CIO losses*". Jan. 16, 2013.

[55] EuSpRiG. "*EuSpRiG Horror Stories*". URL: http://eusprig.org/horror-stories.htm (visited on 06/14/2016).

[56] Felienne Hermans **et al.** "*Data Clone Detection and Visualization in Spreadsheets*". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 292–301. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486827.

[57] S. E. Kruck. "*Testing Spreadsheet Accuracy Theory*". In: *Inf. Softw. Technol.* 48.3 (Mar. 2006), pp. 204–213. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2005.04.005. URL: http://dx.doi.org/10.1016/j.infsof.2005.04.005.

[58] Ruiqing Zhang **et al.** "*How effectively can spreadsheet anomalies be detected: An empirical study*". In: *Journal of Systems and Software* (2016). ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/j.jss.2016.03.061. URL: http://www.sciencedirect.com/science/article/pii/S0164121216300103.

[59] Wensheng Dou, Shing-Chi Cheung **and** Jun Wei. "*Is Spreadsheet Ambiguity Harmful? Detecting and Repairing Spreadsheet Smells due to Ambiguous Computation*". In: *36th International Conference on Software Engineering (ICSE 2014)*. ACM. 2014, pp. 848–858.

[60] Chris Chambers **and** Martin Erwig. "*Automatic Detection of Dimension Errors in Spreadsheets*". In: *J. Vis. Lang. Comput.* 20.4 (Aug. 2009), pp. 269–283. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2009.04.002. URL: http://dx.doi.org/10.1016/j.jvlc.2009.04.002.

[61] Wensheng Dou **et al.** "*Detecting Table Clones and Smells in Spreadsheets*". In: *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2016)*. FSE 2016. Seattle, WA, USA: ACM, 2016, pp. 787–798. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950359. URL: http://doi.acm.org/10.1145/2950290.2950359.

[62] Shing-Chi Cheung **et al.** "*CUSTODES: Automatic Spreadsheet Cell Clustering and Smell Detection Using Strong and Weak Features*". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. New York, NY, USA: ACM, 2016, pp. 464–475. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884796. URL: http://doi.acm.org/10.1145/2884781.2884796.

[63] Felienne Hermans **et al.** "*Data Clone Detection and Visualization in Spreadsheets*". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 292–301. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486827.

[64] Tie Cheng **and** Xavier Rival. In: *Static Analysis: 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*. Ed. by Antoine Miné **and** David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Chap. An Abstract Domain to Infer Types over Zones in Spreadsheets, pp. 94–110. ISBN: 978-3-642-33125-1. DOI: 10.1007/978-3-642-33125-1_9. URL: http://dx.doi.org/10.1007/978-3-642-33125-1_9.

[65] M. Burnett **and** M. Erwig. "*Visually customizing inference rules about apples and oranges*". In: *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*. 2002, pp. 140–148. DOI: 10.1109/HCC.2002.1046366.

[66] Chris Chambers **and** Martin Erwig. "*Reasoning about spreadsheets with labels and dimensions*". In: *Journal of Visual Languages & Computing* 21.5 (2010). Part Special issue on selected papers from VL/HCC'09, pp. 249–262. ISSN: 1045-926X. DOI: http://dx.doi.org/10.1016/j.jvlc.2010.08.004. URL: http://www.sciencedirect.com/science/article/pii/S1045926X10000455.

[67] C. Chambers **and** M. Erwig. "*Dimension inference in spreadsheets*". In: *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. Sept.

2008, pp. 123–130. DOI: 10.1109/VLHCC.2008.4639072.

[68] Michael J Coblenz, Andrew Jensen Ko **and** Brad A Myers. "*Using objects of measurement to detect spreadsheet errors*". In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE. 2005, pp. 314–316.

[69] R. Abraham **and** M. Erwig. "*GoalDebug: A Spreadsheet Debugger for End Users*". In: *29th International Conference on Software Engineering (ICSE'07)*. May 2007, pp. 251–260. DOI: 10.1109/ICSE.2007.39.

[70] Roger Ferguson **and** Bogdan Korel. "*The Chaining Approach for Software Test Data Generation*". In: *ACM Trans. Softw. Eng. Methodol.* 5.1 (Jan. 1996), pp. 63–86. ISSN: 1049-331X. DOI: 10.1145/226155.226158. URL: http://doi.acm.org/10.1145/226155.226158.

[71] Marc Fisher **et al.** "*Automated Test Case Generation for Spreadsheets*". In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. New York, NY, USA: ACM, 2002, pp. 141–153. ISBN: 1-58113-472-X. DOI: 10.1145/581339.581359. URL: http://doi.acm.org/10.1145/581339.581359.

[72] "*Refactoring: Improving the Design of Existing Code*". Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2.

[73] Susmit Jha **et al.** "*Oracle-guided Component-based Program Synthesis*". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. New York, NY, USA: ACM, 2010, pp. 215–224. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806833. URL: http://doi.acm.org/10.1145/1806799.1806833.

[74] Marc Fisher **and** Gregg Rothermel. "*The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms*". In: *Proceedings of the First Workshop on End-user Software Engineering*. WEUSE I. New York, NY, USA: ACM, 2005, pp. 1–5. ISBN: 1-59593-131-7. DOI: 10.1145/1082983.1083242. URL: http://doi.acm.org/10.1145/1082983.1083242.

[75] Wensheng Dou **et al.** "*CACheck: Detecting and Repairing Cell Arrays in Spreadsheets*". In: *IEEE Transactions on Software Engineering (TSE)* (2016).

[76] Felienne Hermans **and** Emerson Murphy-Hill. "*Enron's Spreadsheets and Related Emails: A Dataset and Analysis*". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 7–16. URL: http://dl.acm.org/citation.cfm?id=2819009.2819013.

[77] R. Abraham **and** M. Erwig. "*Header and Unit Inference for Spreadsheets Through Spatial Analyses*". In: *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. Sept. 2004, pp. 165–172. DOI: 10.1109/VLHCC.2004.29.

[78] Felienne Hermans, Martin Pinzger **and** Arie **van** Deursen. "*Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams*". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. New York, NY, USA: ACM, 2011, pp. 451–460. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985855. URL: http://doi.acm.org/10.1145/1985793.1985855.

[79] Gregor Engels **and** Martin Erwig. "*ClassSheets: Automatic Generation of Spreadsheet Applications from Object-oriented Specifications*". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: ACM, 2005, pp. 124–133. ISBN: 1-58113-993-4. DOI: 10.1145/1101908.1101929. URL: http://doi.acm.org/10.1145/1101908.1101929.

[80] Andrew Kennedy. "*Programming languages and dimensions*". 391. University of Cambridge, Computer Laboratory, 1996.

[81] BIPM: Bureau International **des** Poids et Mesures. "*BIPM: Bureau International des Poids et Mesures*". URL: http://www.bipm.org/en/measurement-units/ (visited on 06/05/2016).

[82] Gordon Filby. "*Spreadsheets in Science and Engineering*". Springer, 1995.

[83] Gregg Rothermel **et al.** "*A Methodology for Testing Spreadsheets*". In: *ACM Trans. Softw. Eng. Methodol.* 10.1 (Jan. 2001), pp. 110–147. ISSN: 1049-331X. DOI: 10.1145/366378.366385. URL: http://doi.acm.org/10.1145/366378.366385.

[84] R. Abraham **and** M. Erwig. "*AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets*". In: *Visual Languages and Human-Centric Computing (VL/HCC'06)*. Sept. 2006, pp. 43–50. DOI: 10.1109/VLHCC.2006.11.

[85] Koen Claessen **and** John Hughes. "*QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*". In: *SIGPLAN Not.* 46.4 (May 2011), pp. 53–64. ISSN: 0362-1340. DOI: 10.1145/1988042.1988046. URL: http://doi.acm.org/10.1145/1988042.1988046.

[86] R. Abraham **and** M. Erwig. "*Goal-directed debugging of spreadsheets*". In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. Sept. 2005, pp. 37–44. DOI: 10.1109/VLHCC.2005.42.

[87] The GNOME Project. "*Gnumeric tests*". URL: https://git.gnome.org/browse/gnumeric/tree/test (visited on 06/02/2016).

[88] Alan Blackwell **and** Thomas Green. "*Notational systems–the cognitive dimensions of notations framework*". In: *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science. Morgan Kaufmann* (2003).

[89] T.R.G. Green **and** M. Petre. "*Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework*". In: *Journal of Visual Languages & Computing* 7.2 (1996), pp. 131–174. ISSN: 1045-926X. DOI: http://dx.doi.org/10.1006/jvlc.1996.0009. URL: http://www.sciencedirect.com/science/article/pii/S1045926X96900099.

[90] A. F. Blackwell. "*First steps in programming: a rationale for attention investment models*". In: *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*. 2002, pp. 2–10. DOI: 10.1109/HCC.2002.1046334.

[91] Peter Sestoft **and** Jens Zeilund. "*Sheet-defined functions: implementation and initial evaluation*". Report. Version 1.1. IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark, Jan. 16, 2013.

[92] Microsoft. "*ACC: Visual/Access Basic Is Both a Compiler and an Interpreter*". URL: https://support.microsoft.com/da-dk/kb/109382 (visited on 06/28/2016).

[93] Lee Benfield. "*FMD: Functional Development in Excel*". In: *Proceedings of the 2009 Video Workshop on Commercial Users of Functional Programming: Functional Programming As a Means, Not an End*. CUFP '09. New York, NY, USA: ACM, 2009. ISBN: 978-1-60558-943-5. DOI: 10.1145/1668113.1668121. URL: http://doi.acm.org/10.1145/1668113.1668121.

[94] Rinus Plasmeijer, Marko **van** Eekelen **and** John **van** Groningen. "*Clean Language Report*". Version 2.2. Dec. 2011.

[95] Felienne Hermans, Martin Pinzger **and** Arie **van** Deursen. "*Breviz: Visualizing spreadsheets using dataflow diagrams*". In: *arXiv preprint arXiv:1111.6895* (2011).

[96] Chris Ashby **and** Fraser Atkins. "*SLATE*". URL: www.useslate.com (visited on 06/21/2016).

[97] Florian Biermann. "*Declarative Parallel Programming in Spreadsheet End-User Development: A Literature Review*". Tech. rep. IT University of Copenhagen.

[98] Margaret Burnett. "*What Is End-User Software Engineering and Why Does It Matter?*" In: *End-User Development: 2nd International Symposium, IS-EUD 2009, Siegen, Germany, March 2-4, 2009. Proceedings*. Ed. by Volkmar Pipek **et al.** Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 15–28. ISBN: 978-3-642-00427-8. DOI: 10.1007/978-3-642-00427-8_2. URL: http://dx.doi.org/10.1007/978-3-642-00427-8_2.

[99] Margaret Burnett, Curtis Cook **and** Gregg Rothermel. "*End-user Software Engineering*". In: *Commun. ACM* 47.9 (Sept. 2004), pp. 53–58. ISSN: 0001-0782. DOI: 10.1145/1015864.1015889. URL: http://doi.acm.org/10.1145/1015864.1015889.

[100] Andrew J. Ko **and** Brad A. Myers. "*Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior*". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '04. New York, NY, USA: ACM, 2004, pp. 151–158. ISBN: 1-58113-702-8. DOI: 10.1145/985692.985712. URL: http://doi.acm.org/10.1145/985692.985712.

Appendix

	Sheet Representation	Recalculation	Parallel Recalculation	Templates	Bugs	Smells	Data Clones	Type Systems	Units, Labels and Dimensions	Sheet-defined Functions	Visualisation	Testing and Debugging	End-user Programming
Abraham [69]												✓	
Abraham [14]								✓					
Abraham [11]								✓					
Abraham [46]				✓									
Abraham [84]												✓	
Abraham [86]												✓	
Abraham [77]									✓				
Abraham [47]				✓									
Abramson [38]			✓										
Ahmad [53]								✓					
Ashby [96]											✓		
Benfield [93]										✓			
Burnett [51]												✓	
Burnett [65]									✓		✓		
Burnett [98]													✓
Burnett [99]													✓
Chambers [66]									✓				
Chambers [67]									✓				
Chambers [60]									✓				
Cheng [23]								✓					
Coblenz [68]									✓				
Dou [61]							✓						
Dou [75]						✓							
Dou [59]						✓							
Engels [79]				✓									
Erwig [16]									✓				
Erwig [45]				✓									
Ferguson [70]												✓	
Fisher [71]												✓	
Flemberg [25]			✓										
Hermans [10]						✓					✓		
Hermans [95]											✓		
Hermans [56]							✓				✓		
Hermans [78]											✓		
Hoon [22]										✓			
Nadiminti [39]			✓										
Peyton [15]										✓			

	Sheet Representation	Recalculation	Parallel Recalculation	Templates	Bugs	Smells	Data Clones	Type Systems	Units, Labels and Dimensions	Sheet-defined Functions	Visualisation	Testing and Debugging	End-user Programming
Panko [48]†					✓								
Panko [49]†					✓								
Powell [50]†					✓								
Reichwein [52]												✓	
Rothermel [21]												✓	
Scaffidi [1]													✓
Serek [24]		✓											
Sestoft [2]	✓	✓						✓		✓			✓
Zhang [58]†					✓								

Table 1: An overview of the surveyed literature and their respective categorisation within spreadsheet technology, sorted by authors. Not all literature has an entry in this table. For example, [55] is a list of reports of erroneous spreadsheets on the EuSpRiG website, but does not constitute a paper that examines a particular aspect of spreadsheets per se. The † symbol signifies that the paper is a survey or provides an overview of some subject.