2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Spreadsheets are Code

An Overview of Software Engineering Approaches applied to Spreadsheets

Felienne Hermans, Bas Jansen, Sohon Roy, Efthimia Aivaloglou, Alaaeddin Swidan and David Hoepelman
{f.f.j.hermans, b.jansen, s.roy-1, e.aivaloglou, alaaeddin.swidan}@tudelft.nl, d.j.hoepelman@student.tudelft.nl
Delft University of Technology
The Netherlands

***Abstract*—Spreadsheets can be considered to be the world's most successful end-user programming language. In fact, one could say spreadsheets *are* programs. This paper starts with a comparison of spreadsheets to software: spreadsheets are similar in terms of applications domains, expressive power and maintainability problems. We then reflect upon what makes spreadsheets successful: liveness, directness and an easy deployment environment seem contribute largely to their success.**

Being a programming language, several techniques from software engineering can be applied to spreadsheets. We present an overview of such research directions, including spreadsheet testing, reverse engineering, smell detection, clone detection and refactoring. Finally, open challenges and future plans for the domain of spreadsheet software engineering are presented.

I. Introduction

In addition to professional programmers who are employed to build, maintain and test software, there is also a large community of people programming not as a job, but as a means to an end. These workers, often called *end-user programmers* write queries, small scripts or spreadsheets to support their daily jobs. The number of end-user programmers in the USA alone is conservatively estimated at 11 million compared to only 2.75 million professional programmers [1]. Among end-user programmers, spreadsheets are especially popular. These are used for a large variety of different tasks, from scheduling to financial reporting and from investment analysis to corporate budgeting in all sorts of domains, from small shops to multinationals.

Especially in the financial domain, spreadsheets are ubiquitous. In 2004, the International Data Corporation interviewed 118 business leaders and found that 85% were using spreadsheets in financial reporting and forecasting [2]. Financial intelligence firm CODA reported in 2008 that 95% of all U.S. companies use spreadsheets for financial reporting [2]. In a survey held in 2003 by the US Bureau of Labor Statistics [3], over 60% of 77 million surveyed workers in the US reported using spreadsheets, making this the third most common use of computers, after email and word processing. A more recent survey among 95 companies world-wide, placed spreadsheets fourth, after email, browsing and word processing, accounting for 7.4% of computer time [4]. The Dutch Bureau of Statistics investigates computer literacy among Dutch civilians every year, and has reported a rise in people able to use formulas in spreadsheets from 44% in 2006 to 54% in 2013 [5].

As artifacts of end-user programming, spreadsheets often play a role similar to source code in many companies: they support important organizational processes and often business decisions are taken based on the information calculated and presented in spreadsheets [6].

While spreadsheets are commonly used, their users often have little training as programmers. In spite of that, they often face many of the challenges of professional developers, like choosing which functions to use [7], or understanding someone else's code [8]. Since spreadsheets, like software, frequently contain errors [9], end-users test, verify and debug their programs [10], [11].

These issues that end-users face—issues of program construction, maintenance, testing and debugging—have been topics of research in the programming and software engineering community for decades. Given the similarities between spreadsheets and source code, it is feasible to transform software engineering methods, tools and techniques to make them applicable to spreadsheets. This exactly has been the approach of a number of researchers over the last decade. This paper highlights their past achievements, challenges and future research directions.

II. End-user programming

End-user programming has been a topic of study for decades, mainly started by Nardi [12] in her investigations into spreadsheet use in office workplaces. According to Nardi, the difference between an end-user programmer and a professional programmer lies in their goals. It is the responsibility of a professional developer to build, debug, maintain and sometimes test software for others to use, while end-user developers create programs to support their own domain of expertise, like teaching, planning or bookkeeping [8]. As such, programs that end-users create are, by definition, not meant for others to use, while professional programming has the goal of producing code for others to use.

One of the core problematic aspects of end-user programming is that sometimes the created artifacts evolve from personal solutions to programs used by many colleagues. When that happens, an end-user suddenly, often unintended and unprepared, has to take on challenges of professional developers, like testing, maintaining and generalizing their creations.

III. Spreadsheets are Code

While end-user programming takes on many forms, and their users can be as diverse as system administrators [13] or interaction designers [7], [14], spreadsheets can be considered to be the most successful form of end-user programming.

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.86

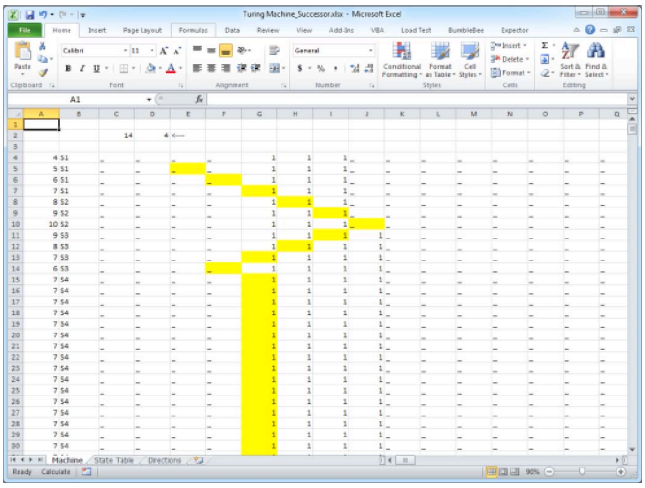


Fig. 1. A Turing machine implementation in Excel, using only formulas. Rows represents consecutive states of the machine and the cells in yellow indicate its head.

Spreadsheets are crucial tools for many workers, enabling millions of users to do reporting, planning, scheduling and all else needed to succeed in their jobs. Of course, not all spreadsheets are full-fledged applications: some are simply used for word processing with an easier layout and do not even contain formulas. Spreadsheets however are also commonly used to create applications, or to perform business critical calculations. We argue this second group of spreadsheets can and should be seen as source code, paving the way to apply methods from software engineering to them.

In the following paragraphs, we outline three reasons why spreadsheets systems are programming languages, and spreadsheets are code.

A. Similar Goals

Firstly, spreadsheets are used for very similar problems, like financial calculations or data manipulation. In such situations, spreadsheets play a role that can also be played by software.

In many cases, the end-users have investigated alternatives, for example, the use of 'off the shelf' solutions for their data or analysis needs. However, these are often expensive and do not fit their needs exactly. A second alternative could be custom made software, either by software engineers within the company or by a contractor. These projects unfortunately have the tendency to go over budget, be delayed and sometimes have disappointing results. Weighting all options: expensive off the shelf product, unsatisfying tailor made software, making a spreadsheet themselves often seems a non-expensive and simple solution for many end-users.

B. Similar Expressive Power

Secondly, spreadsheets have quite some expressive power. In fact, spreadsheet formulas are Turing complete, even without taking Visual Basic for Applications code into account. Using formulas only, you can construct a Turing machine, see Figure 1 [15].

In Excel, it is not possible to continuously update cells—unless some scripting is used such as VBA—so moving the head is mimicked by using one row in the spreadsheet to represent one state of the tape. Each following line represents the state of the tape after one transition, and the head of the machine is visualized with conditional formatting.

C. Similar Maintainability Issues

Finally spreadsheets suffer from typical software problems, including, but not limited to the issues below.

Long life span In some cases, spreadsheets are specifically created for one time use, but more often they stay 'alive': enhanced with more data, reused for next year's budget or modified for a different department. Our research shows that spreadsheets have an average lifespan of five years [6].

Many different users During this long lifespan, spreadsheets are frequently shared among coworkers. On average, twelve different people work with one spreadsheet, performing a variety of tasks on them, including data entry, error checking and analysis [6].

Lack of documentation We found that only one in three spreadsheets contain documentation. We considered even a basic manual to be documentation and did not limit the definition to real technical documentation with design decisions. And still, two thirds of spreadsheets did not contain any explanation [6].

Quality issues Finally, like software, there have been many accounts of big impact errors involving spreadsheets. From somewhat silly errors, like an overbooked Olympic stadium [16], to career wrecking data analysis mistakes [17], the stories of errors are numerous. The European Risk Interest Group keeps a list of these spreadsheet horror stories on their website[1].

IV. Reflections on Spreadsheet Success Factors

Given the wide adoption of spreadsheets, one could wonder why spreadsheets are as successful as they are. In this section we list three reasons contributing to their success.

A. Live Programming

First proposed as a design principle by Maloney and Smith [19], liveness indicates that a user interface is always active and reactive. According to [19] in a live interface *"objects respond to user actions, animations run, layout happens, and information displays are updated continuously"*. More recently live programming has found its way to the public eye, among others by Bret Victor in his talk 'Inventing on Principle' [18]. Figure 2, taken from Victor's talk, illustrates the idea of live programming: on the right, we have source code and on the left, we have the result of that code, in this case: a tree. Modifying the code will immediately affect the tree.

This liveness is also present in spreadsheet systems. When a users enters a formula and presses enter, they see the result, without any effort such as compilation. Liveness of spreadsheets powers their flexibility, often praised as their key success factor.

[1] www.eusprig.org/horror-stories

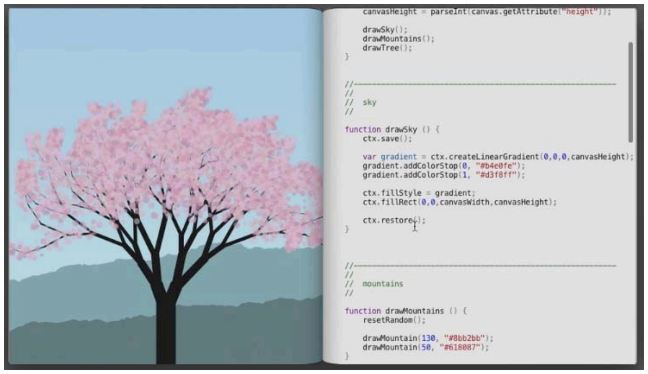
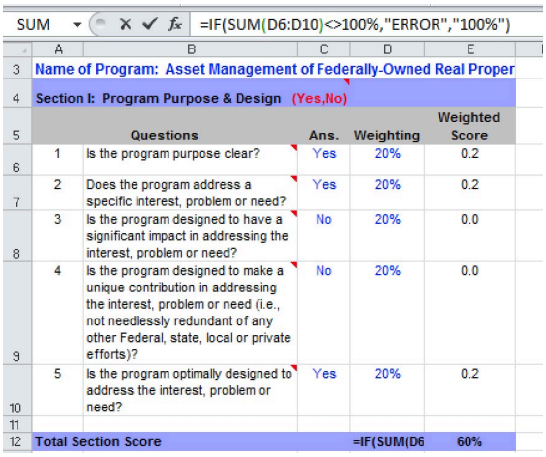


Fig. 2. Live programming: on the right the source code and on the left its instantiation of the code which changes immediately when the code is updated, screenshot from [18].



SUM =IF(SUM(D6:D10)<>100%,"ERROR","100%")

	A	B	C	D	E
3	Name of Program: Asset Management of Federally-Owned Real Proper				
4	Section I: Program Purpose & Design (Yes,No)				
5		Questions	Ans.	Weighting	Weighted Score
6	1	Is the program purpose clear?	Yes	20%	0.2
7	2	Does the program address a specific interest, problem or need?	Yes	20%	0.2
8	3	Is the program designed to have a significant impact in addressing the interest, problem or need?	No	20%	0.0
9	4	Is the program designed to make a unique contribution in addressing the interest, problem or need (i.e., not needlessly redundant of any other Federal, state, local or private efforts)?	No	20%	0.0
10	5	Is the program optimally designed to address the interest, problem or need?	Yes	20%	0.2
11					
12	Total Section Score			=IF(SUM(D6	60%

Fig. 3. A spreadsheet with a test formula in cell D12 expressing that the SUM of the values in D6:D10 should be equal to 100%

B. Directness

Another benefit of spreadsheets is that their interface combines data, metadata and calculations together in one view, and provides the user with easy access to all. Just by clicking a formula, one can manipulate it. This is often called 'directness': *"the feeling that one is directly manipulating the object"* [20]. From a cognitive perspective, directness in computing means *"a small distance between a goal and the actions required of the user to achieve the goal"*[21].

Maloney and Smith describe directness as the fact that a user can *"can initiate the process of examining or changing the attributes, structure, and behavior of user interface components by pointing at their graphical representations directly, as opposed to navigating through an alternate representation."* [19]

This almost exactly describes the interface of a spreadsheet. Instead of navigating to a code behind, a class or an object, spreadsheet users have all ingredients: data, metadata and calculations, in one view, and can access them with one click.

We have often wondered why more advanced spreadsheet users did not use a database or source code, which would have enforced a more structured approach. In our experience, spreadsheet users felt that the distance between the metadata, data and analysis—tables, data and queries—was too big, posing a cognitive load too high.

C. One-click deployment

Another problem which end-user programmers in company settings face is the problem of deployment. Some more advanced users write, for example, Python scripts to analyze data, but are faced with the question of how to get that to run on their neighbor's workstation, with a slightly different version of the operating system, a newer Office and different language settings? Spreadsheets are so universal that almost everyone has a spreadsheet program on their machine. With that, a spreadsheet becomes an executable package with data and calculations packed together, that can run anywhere a spreadsheet system is installed.

V. Achievements

As said, a core approach of research into end-user programming has been to transfer methods from software engineering to end-user languages, and this has been eagerly applied in spreadsheet research. This section presents an overview of successful research directions following this strategy.

A. Testing

One of the programming concepts that found its way to spreadsheets the earliest is testing, with 'What You See Is What You Test' (WYSIWYT) by Rothermel *et al.* [22]. In this paradigm, spreadsheet users have to mark formula outcomes are correct or incorrect, after which the WYSIWYT system calculates which formulas led to the checked values and increases their testedness.

An evaluation of WYSIWYT showed that their approach had an average fault detection percentage of 81% which is *"comparable to those achieved by analogous techniques for testing imperative programs."* [23]. Other studies have confirmed the applicability of testing to spreadsheets [24]. Related is the elegant work of Burnett on spreadsheet assertions that allows spreadsheet users to define assertions and propagates them through the cell dependencies using a similar propagation system [25].

An downside of the WYSIWYT approach is that it requires an annotation of right and wrong values as input, meaning extra effort for the user. When inspecting spreadsheets from practice, we noticed that spreadsheet users often already add simple tests to their spreadsheets expressed with formulas. An example of such a spreadsheet is shown in Figure 3. In the EUSES corpus [26], 9.8% of formulas are such test formulas, which we deemed common enough to be exploited. Hence, we built a tool called Expector that can detect these formulas, store them in a test suite and subsequently calculate coverage and run the tests [10].

B. Reverse Engineering

Like software, spreadsheets often suffer from a lack of documentation. In a field study we found, only one in three

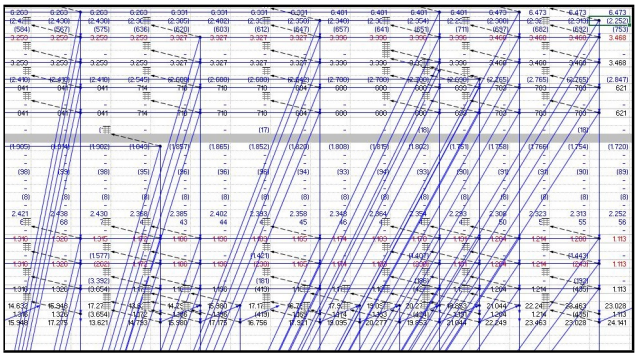


Fig. 4. Spreadsheet shown in Microsoft Excel with the 'trace dependents' option enabled. As can be seen in this picture, this feature does not always support easy understanding.

spreadsheets contained documentation [6]. Despite this lack of documentation, users do often have to understand spreadsheets created by someone else [8], [6]. An approach that was therefore explored by several researchers is to reverse engineer existing spreadsheets in some way.

1) Extracting class diagrams: There have been a number of approaches to extract class diagrams from spreadsheets. We described a method to do so by observing that spreadsheets typically contain three types of data: actual data organized in groups, computations over these groups, and dependencies between them, closely resembling object oriented systems with classes, methods and dependencies [27]. Based on this observation, we implemented a method to transform spreadsheets to class diagrams and evaluated it on the EUSES corpus [26]. We found that we were able to extract diagrams similar to a manually created benchmark in 40% of spreadsheets.

Following this work, Cunha *et al.* [28] described an approach to infer ClassSheets models [29] from spreadsheets, by detecting and exploiting functional dependencies. Their method was validated in a method similar to ours. They manually extracted tables from 27 spreadsheet examples taken from [30]. They found that their method was able to detect correct ClassSheets in about 70% of the cases.

Specifically focusing on the spreadsheets made by scientists, de Vos *et al.* [31] have designed a methodology to extract ontologies in the form of class diagrams from spreadsheets. While their described method is currently manual, they state it could be automated too.

2) Dataflow visualization: In addition to approaches to extract data models as documentation from spreadsheets, there have also been a number of approaches aimed at extracting and visualizing dependencies between cells. One could say that spreadsheets consist of two layers: *visual*: the way in which cells are organized into rows, columns, data blocks and worksheets, and *dependency*: the way in which cells refer to each other.

Firstly, Excel itself contains a feature to overlay a spreadsheet with a dependency graph, which, unfortunately becomes incomprehensible for large and complex spreadsheets, as shown in Fig 4. Observing that this feature did not always support spreadsheet users, we studied the needs that industrial

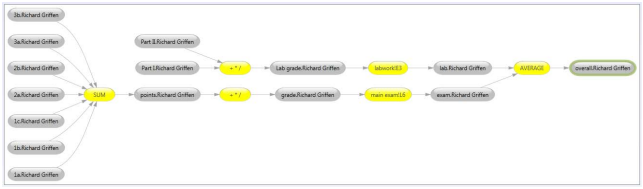


Fig. 5. Formula view of the Leveled Dataflow Visualization as presented in [6]

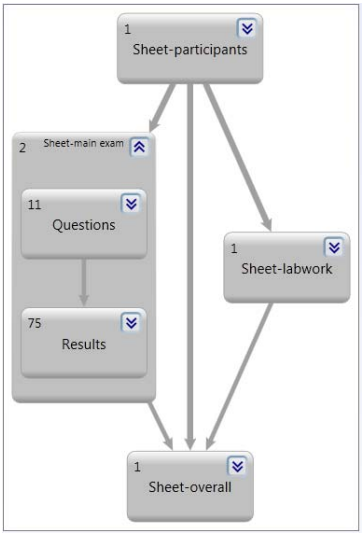


Fig. 6. Worksheet view of the Leveled Dataflow Visualization as presented in [6]

spreadsheet users have when working with spreadsheets [6]. The results confirmed that the most important information needs of spreadsheet users concern the structure of formula dependencies. To meet this demand we developed an approach for the automatic generation of *leveled* data-flow diagrams from spreadsheets. The diagrams are organized in levels and are thus able to represent larger spreadsheets without becoming cluttered, as shown in Figure 6. We implemented the approach and evaluated it with a group of 27 users at Robeco, a Dutch financial services company. The results indicated that end-users consider the tool helpful; the visualizations help them to create story lines for explaining spreadsheets to colleagues, and they scale well to large and complex spreadsheets.

There have been several related efforts to support spreadsheet users in comprehension. Igarashi *et al.* developed a fluid visualization technique based on overlaid animation for better understanding of spreadsheets [32]. Shiozawa *et al.* proposed a technique of cell dependence visualization in 3D based on an interactive lifting up operation [33]. Ballinger *et al.* developed a visualization toolkit that could ease understanding of spreadsheets through visual abstractions in the form of images that emphasize on layout and dependency rather than values of cells [34]. The toolkit was successfully tested on a corpus of 259 spreadsheets but there was no information about the source or size of the spreadsheets.

In addition to these generic works on dataflow visual-

```
=(IF(A7<='Flags(mth)'!$D$21;0;IF(A7<=Inputs!$E$19;IF(A7<Inputs!$E$19;(
(A7-MIN(A6;Inputs!$C$13))/(Inputs!$D$55-Inputs!$C$13))*LOOKUP(
Inputs!$D$55;Notes!$E$8:$CB$8;Notes!$E$553:$CB$553)+C6;IF(A7=
Inputs!$E$19;((A7-MAX(A6;Inputs!$C$13))/(Inputs!$D$55-Inputs!$C$13))
*LOOKUP(Inputs!$D$55;Notes!$E$8:$CB$8;Notes!$E$553:$CB$553);IF(
Repayment!A7=Inputs!$E$19;OFFSET(Notes!$I$553;0;Repayment!#REF!);
IF(A7=Inputs!$D$61;OFFSET(Notes!$I$553;0;Repayment!#REF!);IF(A7>
Inputs!$D$61;0;OFFSET(Notes!$I$553;0;Repayment!#REF!)/2)))))))*
1000000
```

Fig. 7. Formula suffering from the Conditional Complexity smell, inhibiting easy readability

ization, there is also work specifically tailored towards the spreadsheets used by scientists. De Vos *et al.* [35] for example, propose a semi automatic method to infer the calculation workflow underlying a set of spreadsheets. The starting point of their methodology is, like in our approach, the cell dependency graph. The difference is that 1) De Vos *et al.* automatically aggregate all cells in the graph that represent instances and duplicates of the same quantities, based on analysis of the formula syntax. The method also needs an ontology of the spreadsheets domain as input, which is then used to prune the data flow graph by selecting only relevant nodes. They have performed three case studies showing that their generated calculation models approximate the ground truth calculation workflows, both in terms of content and size, but are not a perfect match.

C. Smell Detection

While the work on reverse engineering certainly proves useful when users want to understand or migrate their spreadsheets, sometimes users still got lost in unnecessarily complex formulas. This, again, resembled the problems in source code. Therefore we have coined the idea of *spreadsheet smells*, taking the code smells metaphor of Fowler as a source of inspiration [36].

1) Formula-level Smell Detection: We started by defining smells within individual formulas and we found that many smells defined for code also applied nicely to spreadsheets [37]. As an example, consider conditional complexity. Most spreadsheet systems contain IF and other conditional formulas, so spreadsheet formulas too can suffer from conditional complexity, when many conditionals are nested within one formula. An example of a formula suffering from the *conditional complexity* smell is shown in Figure 7.

Other code smells need small modifications to be applicable to spreadsheets: the *many parameters* smell becomes *many references*: a formula that references a long list of different ranges in a spreadsheet is as smelly as a method with lots of parameters, and the *long method* smell becomes *multiple operations*: a formula that uses a large number of different functions can be hard to understand.

After defining the smells for spreadsheets, we performed an empirical evaluation in which we found that smells occurred within 42.7% of spreadsheets in the EUSES corpus [26] and that *Multiple References* is the most common. We also evaluated the smells with 10 spreadsheets and their users in practice, and found two actual faults with the *duplication*. Also, spreadsheet users agreed that the smells revealed which of the formulas were least maintainable.

More recently, we compared two datasets: one containing spreadsheets which users found unmaintainable, and a version of the same spreadsheets rebuilt by professional spreadsheet developers. The results show that the improved versions suffered from smells to a lesser degree, increasing our confidence that presence of smells indeed coincides with users finding spreadsheets hard to maintain [38].

2) Data Smell Detection: Cunha *et al.* studied smells in spreadsheets too, however they focus on smells in the data, such as cells that do not follow a normal distribution or have a big string distance to other cells in the same region, and thus might be typos. They analyzed 180 spreadsheets from the EUSES corpus, in which they found 3,841 cells suffering from their smells. By manual inspection of the cells, they confirmed that more than 20% of the detected smelly cells point to a possible problem in the spreadsheet [39].

Related work into smells in data was done by Barowy *et al.* who present a tool called CheckCell that identifies cells that have an unusually high impact on the spreadsheet's computations. In an evaluation, the authors showed that CheckCell outperforms standard outlier detection techniques. It successfully finds injected typographical errors produced by a generative model trained with data entry from 169,112 Mechanical Turk tasks [40].

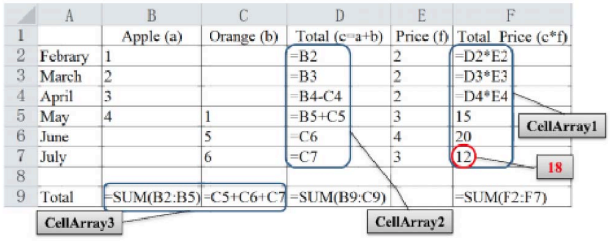
3) Structural Smell Detection: In addition to smells occurring within data or within a single formula, smells can occur in the organization of the spreadsheet, for example when a formula refers to a large number of cells in another worksheet. This is comparable to Fowler's *Feature Envy* smell, where a method of class A uses a large number of fields from class B, and hence can be considered 'envious' of the class B.

We defined three more structural smells in addition to *Feature Envy*, namely: *Inappropriate Intimacy*, *Middle Man* and *Shotgun Surgery* [41]. We again performed a quantitative and qualitative evaluation of our approach, where we first investigated the occurrence of inter-worksheet smells in the EUSES [26] corpus and found that 23.3% of spreadsheets suffered from at least on of the smells, with *Feature Envy* being the most common smell. We also conducted a series of ten case studies at the above described company Robeco. In these case studies we found that inter-worksheet smells can indeed reveal weaknesses in spreadsheets and that subjects confirmed their negative impact on maintenance.

D. Clone Detection

Like in source code, clones or 'copy-pasting' occurs in spreadsheets too, although there are important differences. Copy-pasting in spreadsheets is common: spreadsheet users typically mimic abstraction by copying a similar formula down or left over multiple rows or columns. An example of this is given in the lower part of Figure 8, where the formulas in cells D2 and F2 are copied down and the formula in B2 is copied left.

However, some forms of copying are error prone. In our research we have focused on *data clones*: copies made with the 'paste as values' option supported by Excel. We



	A	B	C	D	E	F
1		Apple (a)	Orange (b)	Total (c=a+b)	Price (f)	Total Price (e*f)
2	Febrary	1		=B2	2	=D2*E2
3	March	2		=B3	2	=D3*E3
4	April	3		=B4-C4	2	=D4*E4
5	May	4	1	=B5+C5	3	15
6	June		5	=C6	4	20
7	July		6	=C7	3	12
8						
9	Total	=SUM(B2:B5)	=C5+C6+C7	=SUM(B9:C9)		=SUM(F2:F7)

(a) A spreadsheet with ambiguous computation smells.

	A	B	C	D	E	F
1		Apple (a)	Orange (b)	Total (c=a + b)	Price (f)	Total Price (e*f)
2	Febrary	1		=B2+C2	2	=D2*E2
3	March	2		=B3+C3	2	=D3*E3
4	April	3		=B4+C4	2	=D4*E4
5	May	4	1	=B5+C5	3	=D5*E5
6	June		5	=B6+C6	4	=D6*E6
7	July		6	=B7+C7	3	=D7*E7
8						
9	Total	=SUM(B2:B7)	=SUM(C2:C7)	=SUM(B9:C9)		=SUM(F2:F7)

(b) A spreadsheet without ambiguous computation smells.

Fig. 8. Two versions of a spreadsheet, with and without ambiguous computation, taken from [42]

designed a detection algorithm [43] to help spreadsheet users in finding and refactoring clones, based on existing clone detection algorithms working on source code [44]. In addition to exact clones, our approach also detects *near-miss clones*, those where minor to extensive modifications have been made to the copied fragments [45]. Our approach was validated both quantitatively and qualitatively. Firstly, we analyzed the EUSES corpus [26] to calculate the precision and performance of our algorithm and to understand how often clones occur. Secondly, we performed two case studies: one with a large budget spreadsheet from our own university and a second one for a large Dutch non-profit organization, for which we analyzed 31 business critical spreadsheets.

Dou *et al.* studied clones in spreadsheets too, focusing on the degeneration of clones formulas [42]. They state that spreadsheet formulas are initially often similar in a row or column. When a spreadsheet evolves however, some cells in such a group will be updated, due to ad hoc modifications or undisciplined copy-and-pastes. Dou *et al.* state these cells suffer from the ambiguous computation smell. They find that such smells are common and likely harmful and propose a tool called AmCheck which automatically detects and repairs ambiguous computation smells. They present a case study on the spreadsheets of the EUSES corpus [26], showing that 44.7% of the spreadsheets suffer from at least one kind of ambiguous computation smell. They then randomly selected 700 sampled smelly groups of cells and manually confirmed that 319 (45.6%) of them were true smelly cell arrays.

E. Refactoring

A final direction on which end-user research has focused on—a logical step after research on smells—is refactoring of end-users artifacts.

We defined refactorings corresponding to our smells [37] and applied them to 10 spreadsheets we received from employees at Robeco, demonstrating that a combination of one or more refactorings from our set could relieve smells in 87% of smelly formulas.

Inspired by our work on spreadsheet smells, Badame and Dig [46] developed a tool called RefBook that supported a number of refactorings for spreadsheet formulas including extract column, replace awkward formula, string to dropdown, introduce cell name, and extract literal. While some of their refactorings can relieve known code smells—for example extract column makes formulas shorter, thus addressing the *multiple operations* smell—their refactorings were not directly related to smells. RefBook was evaluated on 28 users in an online experiment showing that RefBook increases spreadsheet programmer productivity, increases the reliability of transformations, and increases the quality of spreadsheets. Furthermore they studied the EUSES corpus to show that their refactorings can be widely applied. For example, 27% of formulas demonstrated some form of duplication, meaning that Extract Column could be applied to simplify the spreadsheet.

After this, we combined the above two approaches in a new spreadsheet refactoring tool called Bumblebee, which allows a formula to be transformed into another by defining a transformation rule. Therefore, BumbleBee is more generic than RefBook, which only supported a fix number of refactorings [47]. However, initially, this approach has the downside that it can only consider one formula, and not the spreadsheet as a whole, meaning some of RefBook's refactorings like Extract Formula and Introduce Cell Name were not supported. This was addressed by the work of Hoepelman [48], which furthermore introduced new refactorings including Inline Formula and Introduce Conditional Aggregate.

F. Conclusion

We conclude that a diverse range of approaches aimed at source code are applicable to end-user programming in general and spreadsheets in particular. From testing to smells, and from refactoring to reverse engineering, software engineering methods transfer well to end-user programming and a broad range of evaluations demonstrate that users benefit from them.

VI. Challenges

Now that we have described the key successes in the application of software engineering to spreadsheets, including testing, reverse engineering, smell detection, clone detection and refactoring, we direct our attention to challenges in researching software engineering methods in spreadsheets.

A. End-user's Perception and Self Perception

One of the key challenges when researching end-user programming is that users do not see themselves as programmers. In one case where we were working with an investment banker, who was almost insulted when we, impressed with a risk management dashboard he built with Excel, called him a programmer. Because end-user programmers do not self-identify as developers, they often are unaware of tools, methods and techniques that could support them in their programming efforts.

The perception of end-user programmers as not being 'real' programmers is not limited to how programmers view themselves, but also to how they are seen by others. Coworkers, especially those themselves trained as professional developers, often fail to recognize and sometimes even belittle their programming efforts, while professional developers in the workplace could offer great support and could ease technology transfer of spreadsheet solutions.

B. Lack of Best Practices

A challenge that follows from the previous one is the lack of standards. Since spreadsheets and their creators are not seen as source code and programmers respectively, they are often outside of the scope of a diverse range of professionalization efforts within companies. Software, but also numerous processes are standardized; spreadsheets on the other hand rarely are. While a number of spreadsheet standards exist[2][3][4], they have not found widespread adoption yet.

This lack of standards means spreadsheets can be created in many different ways, which inhibits easy comprehension and maintenance, but also makes it harder to automatically analyze and process spreadsheets.

C. Lack of Data

Spreadsheets often contain models and calculations of vital importance to companies, and therefore their users are reluctant to share them with researchers. This is a big challenge with industrial research in general, and spreadsheets in specific. Contrary to source code, which developers often share on online platforms like GitHub, spreadsheet users typically do not share theirs.

While there are a number of public datasets available [26], [49], [50], only one of these ([49]) stems from a company. And even then, we lack information about their creation and the maintenance process around them. Process information that is often available for source code repositories, like issues and version control history, are missing from spreadsheets, prohibiting us from deeply understanding the problems with spreadsheets. We have worked together with companies providing us with data [6], [51], [41], [37], [38] enabling us to study spreadsheets *in the wild*. That however came at the price of reduced repeatability, because we were not allowed to share these spreadsheets.

D. Performance of Research Tools

While many spreadsheets are relatively small, spreadsheets can also grow extremely large over their long lifespan. The biggest spreadsheet from the Enron set had no less than 175 worksheets, and about 10% of spreadsheets in the corpus have 10 or more worksheets. Large spreadsheets, especially those with heavy connections between the worksheets are hard for users to understand and maintain, but also seem to hinder adoption of research tools, especially for those aimed at supporting maintenance and comprehension. For example, Igarashi *et al.* reported that their animations degrade for spreadsheets larger than 400 cells [32]. Compared to spreadsheets that are found in the industry, 400 cells is a very low limit. We ourselves have also found that parsing and analyzing large numbers of formulas can be more time-consuming that one would expect. Thus it appears that although these tools typically perform well on laboratory examples, industrial adoption is far away for real-life spreadsheets, as tools get unreasonably slow on large spreadsheets.

E. Proprietary Software

A final challenge that we identify in working with spreadsheets is the fact that the most common spreadsheet system, Microsoft Excel, is proprietary and closed source. Simply accessing the formulas already poses a challenge. While there is a way to programmatically access Excel worksheets through Excel in C# and other languages[5], this method does not scale for big spreadsheets or for batch processing large amounts of them.

This makes analysis hard, especially when there is a need to parse spreadsheet formulas, which is needed for research efforts, including for testing and refactoring approaches. There is a formal specification of the Excel grammar available, but this specification consists of about 30 pages of production rules, making it a demanding effort to reimplement. This resulted in the fact that several researchers had to reverse engineer and approximate the excel formula grammar. Various papers have attempted to cover a small subset. For example, in his thesis, Badame presents a grammar that does not cover all possible formulas [52]. Other papers process formulas, but do not share their parser or grammar, like the CheckCell paper [40] and the works by Cunha *et al.* [28], [39]. We recently released an open source version of the grammar[6] which is capable of parsing 99.9% of formulas in the EUSES and Enron dataset [53].

Another problem is that Excel limits the power of add-ins, most notably by disallowing access to the undo-stack, but also limiting the possibilities to traverse cell dependencies deeper than one level. These both increased our difficulties in developing the refactoring plugin [47].

Obviously, one could avoid such these issues by implementing a whole spreadsheet system from scratch, but there is a trade off between easy extendability and power of tools on the one hand and realism on the other hand. As a researcher you also want to attract a large group of people to be able to try your tools, so there is something to say for building on top of Excel, even in the presence of the above challenges.

VII. Future opportunities

In the previous sections we have described a number directions in which the application of software engineering to spreadsheets has proven to be successful. Given these achievements and the challenges we identified in terms of perception of end-users, best practices, lack of data, size and performance of spreadsheets, we identify the following viable directions for future work in the area of spreadsheet software engineering.

[2] http://www.fast-standard.org/

[3] http://www.ssrb.org/standards

[4] http://www.spreadsheetsafe.com/

[5] https://msdn.microsoft.com/en-us/library/office/Microsoft.Office.Interop.Excel.aspx

[6] https://github.com/spreadsheetlab/XLParser

A. Performance

Understandability and maintainability are not the only issues with large spreadsheets, large spreadsheets can also suffer from serious performance problems.

Some researchers have attempted to improve spreadsheet performance in various ways, for example, Pichitlamken *et a.l* [54] proposed a method to offload a simulation model built in spreadsheets into a grid of computing resources. Their tool, while useful, was developed for a very specific set of conditions, making it difficult to apply to spreadsheets in general. A related effort is the work by Abramson *et al.* in [55] who presented ActiveSheets, a solution designed to evaluate "otherwise sequential" custom functions of Excel spreadsheets, by creating a middle layer that processes the requests for evaluations. Finally, there is ExcelGrid [56], in which a middle ware tool was designed and built connecting an Excel spreadsheet with either a private or a public cluster.

The fact that there is some preliminary research done in this direction, indicates that this is a problem worthy of more research. But, while the above tools and techniques do improve performance in the spreadsheets under study, they do not take the content of the spreadsheet into account, by for example identifying hotspots in the spreadsheets calculation. We believe that by combining High Performance Computing with smell detection and refactoring, tailored spreadsheet improvements could be made.

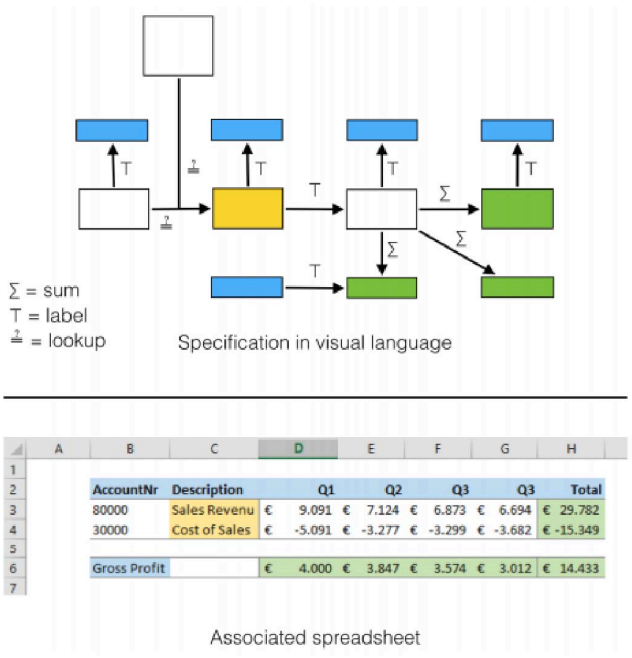
B. Deeply Understanding Spreadsheets in the Enterprise

So far, research on spreadsheets, both by us and by others, has focused on individual spreadsheets and their users. In reality, spreadsheets are often part of a larger end-user ecosystem, where users, for example, import data from a data warehouse, process it in a spreadsheet and then write a report about it in Word. We see the broadening of the scope of end-user programming as a very viable direction for future research.

This research direction will aim to understand why people continue to resort to 'home brew' solutions, while there are software systems in place. What functionality or power do they miss in existing tools? Understanding what drives people to spreadsheets will support us in building supporting or replacing tools.

C. Domain-Specific Spreadsheets

Not all spreadsheets are created equally. While working with spreadsheets in practice we have seen that there are several high-level classifications to be made. For example: some spreadsheets are used for financial modeling and project future revenues and costs along a time-axis, other spreadsheets are used for a calculation on a single point in time, for example cost price calculations. In other cases spreadsheets are used purely for reporting purposes and are lacking any complex calculations. In some spreadsheets the calculations are even missing, the spreadsheet is just a collections of lists and is actually used as a database. We hypothesize that each of those categories of spreadsheets needs a different type of support. For example, there might be 'budget specific' smells or tests, which are only useful to users making a budget, but would not support, for example, a cost price calculation. By analyzing different types of spreadsheets, we could tailor our methods to specialized spreadsheet types.



	A	B	C	D	E	F	G	H
1								
2		AccountNr	Description	Q1	Q2	Q3	Q3	Total
3		80000	Sales Revenu	€ 9.091	€ 7.124	€ 6.873	€ 6.694	€ 29.782
4		30000	Cost of Sales	€ -5.091	€ -3.277	€ -3.299	€ -3.682	€ -15.349
5								
6		Gross Profit		€ 4.000	€ 3.847	€ 3.574	€ 3.012	€ 14.433
7								

Fig. 9. We envision the visual language to look like this. On top there is the model the user would create and on the bottom we find the associated spreadsheet

D. Higher Level of Abstraction

In previous work we have tried to understand what causes spreadsheets to be error prone. Contrary to our initial hypotheses, studies have found that the level of complexity and coupling in spreadsheets is not that large [38]. So one could wonder what it is that makes spreadsheets error-prone and hard to understand. One hypothesis is that the interface of spreadsheets, with all its freedom for users in how to layout spreadsheets, is a double-edged sword. While allowing the users this total freedom, the interface is not supporting users in making the right choices. One solution we envision is a higher level language to create spreadsheets, as illustrated by Figure 9. When we have a definition of the spreadsheet at a higher level of abstraction, we can subsequently help users to select the right formatting and also support them in changing the layout of the spreadsheet later in the process.

E. Beyond Formulas

Spreadsheets themselves are changing, and many vendors, including Microsoft, are trying to enable spreadsheet developers to build more powerful programs, for example with

QlikView[7], Tableau[8] and PowerBI[9]. Currently, these tools are certainly bringing more power to end-users, but their creators are mainly concerned with enabling users to build new analyses and not with helping users maintaining them. We thus hypothesise that this new generation of end-user programming artifacts will again suffer from maintainablilty issues, like an unexpectedly long life span, and will thus be be in need of testing and refactoring.

VIII. Conclusion

This paper presents an overview of research papers that apply software engineering to spreadsheets. We first make the case that spreadsheets are code: they are used for similar problems, have similar expressive power and suffer from similar problems. We then reflect upon the success of spreadsheets, what makes them the world's most used programming language for end-users? We assert their liveness, directness and easy deployment are factors contributing to their widespread adoption.

Because of their similarity to source code, applying methods from software engineering to them has proven a successful research direction for many. In this paper we summarize achievements in the area of testing, reverse engineering, smell detection, clone detection and refactoring. We also highlight challenges in researching spreadsheets, including the perception that spreadsheet developers are not real programmers, the lack of best practices, the lack of readily available data, the difficulties of analyzing large spreadsheets and issues building upon on proprietary software like Excel.

We close this paper by identifying a number of viable research directions, including investigating how to improve the performance of spreadsheets, understanding the process around them, making research more domain specific, raising the level of abstraction for spreadsheet users and finally analyzing reporting artifacts other than spreadsheets.

References

[1] C. Scaffidi, M. Shaw, and B. A. Myers, "Estimating the numbers of end users and end user programmers," in *Proc. of VL/HCC '05*, 2005, pp. 207–214.

[2] R. Panko, "Security and sarbanes oxley: What about the spreadsheets?"

[3] USA Bureau of Labor Statistics. (2005) Computer and internet use at work in 2003. [Online]. Available: http://www.bls.gov/news.release/pdf/ciuaw.pdf

[4] Wellnomics. (2007) An analysis of computer use across 95 organisations in europe, north america and australasia. [Online]. Available: http://wellnomics.com/assets/Uploads/White-Papers/Wellnomics-white-paper-Comparison-of-Computer-Use-across-different-Countries.pdf

[5] Dutch Bureau of Statistics. (2014) Ict gebruik van personen naar persoonskenmerken. [Online]. Available: http://statline.cbs.nl/Statweb/publication/?DM=SLNL&PA=71098ned&D1=23,29&D2=0-2&D3=a&VW=T

[6] F. Hermans, M. Pinzger, and A. van Deursen, "Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 451–460. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985855

[7] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, ser. VLHCC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 199–206. [Online]. Available: http://dx.doi.org/10.1109/VLHCC.2004.47

[8] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 21:1–21:44, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1922649.1922658

[9] R. R. Panko, "What we know about spreadsheet errors," *Journal of End User Computing*, vol. 10, no. 2, pp. 15–21, May 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=287893.287899

[10] F. Hermans, "Improving spreadsheet test practices," in *Center for Advanced Studies on Collaborative Research, CASCON '12, Toronto, ON, Canada, November 18-20, 2013*, 2013, pp. 56–69. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555531

[11] A. J. Ko and B. A. Myers, "Designing the whyline: A debugging interface for asking questions about program behavior," in *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM*, 2004, pp. 151–158.

[12] B. A. Nardi, *A small matter of programming: perspectives on end user computing*. MIT Press, 1993.

[13] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, and M. Prabaker, "Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices," in *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW '04. New York, NY, USA: ACM, 2004, pp. 388–395.

[14] B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko, "How designers design and program interactive behaviors," in *IEEE Symposium on Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008*, pp. 177–184.

[15] F. Hermans. (2013) Excel turing machine. [Online]. Available: http://www.felienne.com/archives/2974

[16] P. Kelso. (2012) London 2012 olympics: lucky few to get 100m final tickets after synchronised swimming was overbooked by 10,000. [Online]. Available: http://www.telegraph.co.uk/sport/olympics/8992490/London-2012-Olympics-lucky-few-to-get-100m-final-tickets-after.-synchronised-swimming-was-overbooked-by-10000.html

[17] T. Herndon, M. Ash, and R. Pollin, "Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff," *Cambridge Journal of Economics*, vol. 38, no. 2, pp. 257–279, 2014. [Online]. Available: http://cje.oxfordjournals.org/content/38/2/257.abstract

[18] B. Victor. (2012) Inventing on principle. Invited talk at the Canadian University Software Engineering Conference (CUSEC). [Online]. Available: http://vimeo.com/36579366

[19] J. H. Maloney and R. B. Smith, "Directness and Liveness in the Morphic User Interface Construction Environment," in *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, ser. UIST '95. New York, NY, USA: ACM, 1995, pp. 21–28. [Online]. Available: http://doi.acm.org/10.1145/215585.215636

[20] B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, vol. 16, no. 8, pp. 57–69, Aug. 1983.

[21] M. M. Burnett, "Visual Programming," in *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, Inc., 2001. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/047134608X.W1707/abstract

[22] G. Rothermel, L. Li, and M. Burnett, "Testing strategies for form-based visual programs," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ser. ISSRE '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 96–. [Online]. Available: http://dl.acm.org/citation.cfm?id=851010.856084

[23] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel, "Wysiwyt testing in the spreadsheet paradigm: an empirical evaluation," in *Proc. of INCSE '00*, 2000, pp. 230–239.

[24] S. E. Kruck, "Testing spreadsheet accuracy theory," *Information & Software Technology*, vol. 48, no. 3, pp. 204–213, 2006.

[7] http://www.qlik.com/products/qlikview

[8] www.tableau.com

[9] https://powerbi.microsoft.com

[25] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace, "End-user software engineering with assertions in the spreadsheet paradigm," in *Proc. of ICSE '03*, 2003, pp. 93–103. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776828

[26] M. Fisher and G. 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*, ser. WEUSE I. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083242

[27] F. Hermans, M. Pinzger, and A. v. Deursen, "Automatically Extracting Class Diagrams from Spreadsheets," in *ECOOP 2010 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, T. DHondt, Ed. Springer Berlin Heidelberg, Jun. 2010, no. 6183, pp. 52–75. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-14107-2_4

[28] J. Cunha, M. Erwig, and J. Saraiva, "Automatically Inferring ClassSheet Models from Spreadsheets," in *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VLHCC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 93–100. [Online]. Available: http://dx.doi.org/10.1109/VLHCC.2010.22

[29] G. Engels and M. Erwig, "ClassSheets: Automatic Generation of Spreadsheet Applications from Object-oriented Specifications," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 124–133. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101929

[30] S. Powell and K. Baker, *The Art of Modeling with Spreadsheets*. Hoboken, N.J: John Wiley & Sons, Inc., 2003.

[31] M. D. Vos, W. R. V. Hage, J. Ros, and G. Schreiber, "G.: Reconstructing Semantics of Scientific Models : a Case Study," in *In: Proceedings of the OEDW workshop on*, 2012.

[32] T. Igarashi, J. D. Mackinlay, B.-W. Chang, and P. T. Zellweger, "Fluid visualization of spreadsheet structures," in *Proceedings. IEEE Symposium on Visual Languages, 1998*. IEEE, 1998, pp. 118–125.

[33] H. Shiozawa, K.-i. Okada, and Y. Matsushita, "3d interactive visualization for inter-cell dependencies of spreadsheets," in *Proceedings. 1999 IEEE Symposium on Information Visualization, 1999.(Info Vis' 99)*. IEEE, 1999, pp. 79–82.

[34] D. Ballinger, R. Biddle, and J. Noble, "Spreadsheet visualisation to improve end-user understanding," in *Proceedings of the Asia-Pacific symposium on Information visualisation-Volume 24*. Australian Computer Society, Inc., 2003, pp. 99–109.

[35] M. de Vos, J. Wielemaker, G. Schreiber, B. Wielinga, and J. Top, "A Methodology for Constructing the Calculation Model of Scientific Spreadsheets," in *Proceedings of the 8th International Conference on Knowledge Capture*, ser. K-CAP 2015. New York, NY, USA: ACM, 2015, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2815833.2815843

[36] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[37] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas," *Empirical Software Engineering*, vol. 20, no. 2, pp. 549–575, 2014. [Online]. Available: http://link.springer.com/article/10.1007/s10664-013-9296-2

[38] B. Jansen and F. Hermans, "Code smells in spreadsheet formulas revisited on an industrial dataset," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME '15)*, 2012, to appear.

[39] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva, "Towards a Catalog of Spreadsheet Smells," in *Computational Science and Its Applications ICCSA 2012*, ser. Lecture Notes in Computer Science, B. Murgante, O. Gervasi, S. Misra, N. Nedjah, A. M. A. C. Rocha, D. Taniar, and B. O. Apduhan, Eds. Springer Berlin Heidelberg, Jun. 2012, no. 7336, pp. 202–216, dOI: 10.1007/978-3-642-31128-4_15. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-31128-4_15

[40] D. W. Barowy, D. Gochev, and E. D. Berger, "CheckCell: Data Debugging for Spreadsheets," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 507–523. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660207

[41] F. Hermans, M. Pinzger, and A. v. Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 441–451. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337275

[42] W. Dou, S.-C. Cheung, and J. Wei, "Is Spreadsheet Ambiguity Harmful? Detecting and Repairing Spreadsheet Smells Due to Ambiguous Computation," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 848–858. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568316

[43] F. Hermans, B. Sedee, M. Pinzger, and A. v. Deursen, "Data clone detection and visualization in spreadsheets," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 292–301. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486827

[44] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proc. of CASCON '93*, 1993, pp. 171–183.

[45] C. K. Roy, "Detection and analysis of near-miss software clones," in *Proc. of ICSM '09*, 2009, pp. 447–450.

[46] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2012, pp. 399–409.

[47] F. Hermans and D. Dig, "BumbleBee: A refactoring environment for spreadsheet formulas," *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE*, pp. 747–750, 2014. [Online]. Available: http://files.figshare.com/1757475/bumblebee.pdf

[48] D. Hoepelman, "Tool-assisted spreadsheet refactoring and parsing spreadsheet formulas," Master's thesis, Delft University of Technology, the Netherlands, 2015.

[49] F. Hermans and E. Murphy-Hill, "Enron's spreadsheets and related emails: A dataset and analysis," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 7–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819013

[50] T. Barik, K. Lubick, J. Smith, J. Slankas, and E. R. Murphy-Hill, "Fuse: A reproducible, extendable, internet-scale corpus of spreadsheets." in *MSR*. IEEE, 2015, pp. 486–489. [Online]. Available: http://dblp.uni-trier.de/db/conf/msr/msr2015.html#BarikLSSM15

[51] F. Hermans, M. Pinzger, and A. V. Deursen, "Detecting Code Smells in Spreadsheet Formulas," *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2012. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6405300

[52] S. Badame, "Refactoring meets spreadsheet formulas," Master's thesis, University of Illinois at Urbana-Champaign, United States of America, 2012.

[53] E. Aivaloglou, D. Hoepelman, and F. Hermans, "A grammar for spreadsheet formulas evaluated on two large datasets," in *15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '15)*, 2015.

[54] J. Pichitlamken, S. Kajkamhaeng, P. Uthayopas, and R. Kaewpuang, "High performance spreadsheet simulation on a desktop grid," *Journal of Simulation*, vol. 5, no. 4, pp. 266–278, 2010.

[55] D. ABRAMSON, P. ROE, L. KOTLER, and D. MATHER, *ActiveSheets Super-Computing with Spreadsheets*, 2001.

[56] K. Nadiminti, Y.-F. Chiu, N. Teoh, A. Luther, S. Venugopal, and R. Buyya, "Excelgrid: A .net plug-in for outsourcing excel spreadsheet workload to enterprise and global grids," 2004.