Combining Smells and Fault Localization in Spreadsheets

Rui Abreu, Jácome Cunha, João Paulo Fernandes,
Pedro Martins, Alexandre Perez, João Saraiva

Abstract

Despite being staggeringly error prone, spreadsheets are a highly flexible programming environment that is widely used in industry. In fact, spreadsheets are widely adopted for decision making, and decisions taken upon wrong (spreadsheet-based) assumptions may have serious economical impacts on businesses, among other consequences.

This paper proposes a technique to automatically pinpoint potential faults in spreadsheets. It combines a catalog of spreadsheet smells that provide a first indication of a potential fault, with a generic spectrum-based fault localization strategy in order to improve (in terms of accuracy and false positive rate) on these initial results. Our technique has been implemented in a tool which helps users detecting faults.

To validate the proposed technique, we consider a well-known and well-documented catalog of faulty spreadsheets. Our experiments yield two main results: *i)* we were able to distinguish between smells that can point to faulty cells from smells and those that are not capable of doing so, and *ii)* we provide a technique capable of detecting a significant number of errors: two thirds of the cells labeled as faulty are in fact (documented) errors.

1 Introduction

Spreadsheet systems are a landmark in the history of generic software products. They have achieved an astonishing success in terms of both the number of users and the variety of domains in which they are nowadays used. Just as an indication, it is estimated that 95% of all U.S. companies use spreadsheets for financial reporting [1], and that 90% of all analysts in the industry perform calculations in spreadsheets [1]. Furthermore, as shown in a recent study performed at an asset management company, 52% of all spreadsheets were used for calculation tasks, and are the basis for decisions within that company [2].

This importance, however, has not been coupled with effective mechanisms for error prevention, as shown by several studies [3, 4], and by a long list of horror stories with huge social and economic impact[1].

[1]This list is available at: `http://www.eusprig.org/horror-stories.htm`

One particularly regrettable example in this list involves Portugal, which currently undergoes a financial rescue plan based on intense austerity whose merit was co-justified upon [5]. The fact is that the conclusions drawn there have been publicly questioned given that a formula range error was found in the spreadsheet supporting the authors' calculations. While the authors have later re-affirmed their original conclusions, the public pressure was so intense that a few weeks later they felt the need to publish an errata of their 2010 paper. It is furthermore unlikely that the concrete social and economical impacts of that particular spreadsheet error will ever be determined.

In practice, all these evidences seem to suggest that spreadsheet error prevention, detection and debugging techniques are much needed. The natural trend of incorporating well-established programming language features under spreadsheets has been witnessed, for example, by the integration of spectrum-based fault localization methods under a spreadsheet system [6] and the identification of spreadsheet *bad smells* [7, 8, 9, 10]. Note that both these techniques are well established for general purpose programming languages: [11] and [12], respectively.

In this paper, we combine bad smell detection and fault localization techniques to create a new debugging framework for spreadsheets. We proceed in three distinct phases.

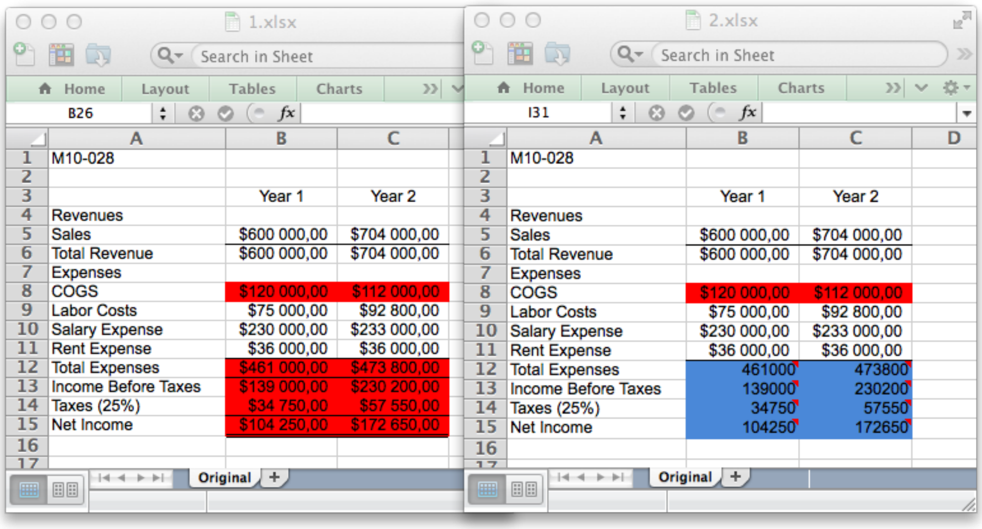
Firstly, we analyze the extensive catalog of spreadsheet smells that has been published in the literature [7, 8, 9, 10]. Indeed, as a smell does not necessarily correspond to an error, we seek to divide this catalog into two: i) the one of *fault-inducing* smells, i.e., the smells that are capable, on their own, of signaling spreadsheet errors and ii) the one of *perceived* smells, that by nature do not contribute to this goal. This analysis relies on the independent spreadsheet corpus [13], which is composed of 73 spreadsheets that contain errors that have previously been documented in detail.

Secondly, the cells that are signaled by fault-inducing smells are provided as input to a fault localization algorithm, that is, such cells are given to the algorithm as potentially being faulty. In this step, we seek to confirm the faulty nature of certain cells, or otherwise discard them as faults. Moreover, we also attempt to identify other spreadsheet cells that may have contributed to the faults identified in the previous cells (e.g. referenced cells may not be marked by the smell algorithms, but may be used in a faulty cell, making them potential faults). In the end of this step our technique has identified all the potential faults in a spreadsheet.

Finally, we have extensively evaluated the debugging method that we propose, again by using the corpus of [13]. Our results show that, on average, two out of three cells identified by our techniques are (documented) errors, and that we are able to locate $\sim$70% of the existing errors.

This paper is an extension of previous work [14], where the combination of spreadsheet smell detection and spectrum-based fault localization was first introduced. It improves our conference submission in the following:

1. We incorporated in our studies, and in our tool, two new smells from the



	A	B	C
1	M10-028		
2			
3		Year 1	Year 2
4	Revenues		
5	Sales	$600 000,00	$704 000,00
6	Total Revenue	$600 000,00	$704 000,00
7	Expenses		
8	COGS	$120 000,00	$112 000,00
9	Labor Costs	$75 000,00	$92 800,00
10	Salary Expense	$230 000,00	$233 000,00
11	Rent Expense	$36 000,00	$36 000,00
12	Total Expenses	$461 000,00	$473 800,00
13	Income Before Taxes	$139 000,00	$230 200,00
14	Taxes (25%)	$34 750,00	$57 550,00
15	Net Income	$104 250,00	$172 650,00
16			
17			

	A	B	C	D
1	M10-028			
2				
3		Year 1	Year 2	
4	Revenues			
5	Sales	$600 000,00	$704 000,00	
6	Total Revenue	$600 000,00	$704 000,00	
7	Expenses			
8	COGS	$120 000,00	$112 000,00	
9	Labor Costs	$75 000,00	$92 800,00	
10	Salary Expense	$230 000,00	$233 000,00	
11	Rent Expense	$36 000,00	$36 000,00	
12	Total Expenses	461000	473800	
13	Income Before Taxes	139000	230200	
14	Taxes (25%)	34750	57550	
15	Net Income	104250	172650	
16				
17				

(a) Spreadsheet with errors. (b) Spreadsheet with faults located.

Figure 1: The same spreadsheet before and after our fault detection technique is applied.

literature; these smells were included since they signal potential problems in spreadsheet cells;

This addition can be seen in **Sections 3 and 5**;

2. We provided a more detailed introduction to the application of spectrum-based fault localization to the spreadsheet domain, supported by a running example explaining the focal steps. Also presented is a sensibility analysis evaluating the diagnostic approach's ability to handle misclassified input;

   This addition can be seen in **Section 4**;

3. We analyzed the error detection nature of these smells, in the line of what we have done in the past with all other smells in our catalog, and classified them as either Fault-inducing or Perceived;

   This addition can be seen in **Section 6**;

4. We included the one that is indeed Fault-inducing in the combination of smells that our technique is based on;

   This addition can be seen in **Section 7**;

5. We validated our technique considering a totally new case study; we have taken a well know, error free, repository of spreadsheets from the literature, with the aim on finding how many false positives our technique signals. We observed numbers of around 1,5%;

This addition can be seen in **Section 8**;

6. We updated the related work section to feature the latest work on spreadsheet debugging.

   This addition can be seen in **Section 9**;

This paper is organized as follows. We start by presenting an example to motivate our approach in Section 2. In Section 3 we briefly review the smells proposed for spreadsheets. We continue in Section 4 introducing the current techniques for spectrum-based fault localization. The tool we have implemented is explained in detail in Section 5. In Section 6 we evaluate how each smell would behave as a fault detector and filter out those that cannot explain any fault. Section 7 details how to use smells as inputs to a spectrum-based fault localization algorithm to find errors in spreadsheets. In Section 9 we discuss related work and finally in Section 10 we draw some conclusions and present future work.

2 Motivation

This section motivates our approach using a spreadsheet taken from the spreadsheet corpus of [13]. This spreadsheet can be seen in Figure 1a and represents the estimated expenses and revenues of a company for two different years, with parameters like labor, rent, and taxes. This spreadsheet has ten cells whose observed values are wrong (marked with red[2] in Figure 1a). As the net income depends on other values that are wrong (since `B15=B13-B13` and `C15=C13-c14`), the ultimate goal of using such a spreadsheet is compromised and will produce incorrect estimations.

In our approach, the first step to find these faulty cells is to apply spreadsheet smells, individually or in combination, and label the cells that are considered smelly (but not necessarily faulty). In a second step of our approach, the cells identified previously as smelly act as input to a fault localization algorithm that points out potential problematic cells. Finally, we signal toxic cells, i.e., cells that work as dependencies of faulty ones. Finally, we point a set of cells that we indicate as being potentially faulty.

Let us consider the spreadsheet of Figure 1a. Just by considering a single spreadsheet smell – *Multiple References*[3] – combined with a fault-localization algorithm allows us to identify the errors marked with blue in the spreadsheet of Figure 1b.

We see that, in this particular case, 8 out of 10 errors were found by our method. Moreover, in this example, our approach did not produce any false positives. That is to say that it did not mark as faulty any correct cell. In this paper we consider a full catalog of spreadsheet smells and a large corpus of

[2]We assume colors are visible on final digital/printed versions of this paper.

[3]This smell appears when a formula references too many cells, reducing its understandability.

documented faulty spreadsheets. Our approach is able to locate more than 70% of the corpus' faulty cells. About 2 out of 3 cells that we identify correspond to a (documented) fault.

3 Spreadsheet Smells

The concept of *code smell* (bad *smell*, or just *smell*) was introduced by Martin Fowler as a first symptom that may correspond to a deeper problem in a system [12]. This means that a smell does not always imply an error. For example, a method with ten parameters, despite being smelly, may be perfectly implemented.

Along with the definition of smell, Martin Fowler also proposed an initial catalog of potential problems in the form of smells. Although this catalog was originally defined for source code, the smells identified in it may sometimes be applied to other artifacts, such as spreadsheets.

Fowler's work inspired several authors to propose different catalogs of smells for spreadsheets. We have taken the union of all the proposed catalogs, obtaining the comprehensive list shown below. The first six smells in this list were proposed in [9, 15], and exploit, for example, statistical properties of spreadsheet data in the same row/column. The following five smells (7 to 11) have appeared in [8], and refer to spreadsheet formulas. The next group of four smells (12 to 15) in this list deals with inter-worksheet smells [7]. Finally, the last two smells (16 and 17) were defined to find problematic cells in regions in spreadsheets [16][4].

Each smell in the following list has a number which will be used later on for identification:

1 – Standard Deviation: This smell detects, for a group of cells holding numerical values, the ones that do not follow their normal distribution.

2 – Empty Cell: Cells that are left empty but that occur in a context that suggests they should have been filled in are detected by this smell.

3 – Pattern Finder: This smell finds patterns in a spreadsheet such as a row containing only numerical values except for one cell holding a label/formula or being empty.

4 – String Distance: Typographical errors are frequent when inputing data. In order to try to detect these type of errors in spreadsheets, this smell signals string cells that differ minimally with respect to other surrounding cells.

5 – Reference to Empty Cells: The existence of formulas pointing to empty cells is a typical source of spreadsheet errors. This smell detects such occurrences.

[4]Actually, this work presents other smells, but some of them are already included in the list we present, and others do not apply to our work; indeed, we rely on cells being signaled by smells, and these mark entire regions of the spreadsheet.

6 – Quasi-Functional Dependencies: In [17] it is described a technique to identify dirty values using a slightly relaxed version of Functional Dependencies (FD) [18]. There exists a FD from a column A to a column B if multiple occurrences of the same value in A always correspond to the same value in B, except for a small number of cases.

7 – Multiple Operations: This smell is inspired by the well-known code smell *Long Method*. As in long methods, formulas with many different operations will likely be hard to understand. This is especially problematic in spreadsheets since in most spreadsheet systems, there is limited space to view a formula, causing long ones to be cut off.

8 – Multiple References: This smell appears when a formula references many different cells, reducing its understandability. An example is: `SUM(A1:A5; B7; C18; C19; F19)`.

9 – Conditional Complexity: As also happens in source code, this smell detects formulas with many conditional operations. For example: `IF(A3=1, IF(A4=1, IF(A5<34700, 50)), 0)`.

10 – Long Calculation Chain: Spreadsheet formulas can create chains of calculations since they can refer to other formulas. To understand the purpose of such formulas, users must trace along multiple steps to find the origin of the data and intermediate calculations.

11 – Duplicated Formulas: This smell indicates that similar snippets of code are used throughout a class. This also happens in spreadsheets since some formulas are partly the same as others. For example, `SUM(A1:A6)+10%` and `SUM(A1:A6)+20%` have the first part duplicated.

12 – Inappropriate Intimacy: This smell was proposed to flag classes with too many dependencies of another class. In spreadsheets this can be adapted to recognize a worksheet that is too much related to a second one.

13 – Feature Envy: This smell appears when a formula is more interested in cells from another worksheet, which suggests it should be moved to it.

14 – Middle Man: A middle man is a class that delegates most of its operations to other classes, and does not contain enough logic to justify its own existence. In spreadsheets this occurs if a 'middle man' formula contains only a reference to other cells, like the formula `=Sheet1!A2`.

15 – Shotgun Surgery: This happens in spreadsheets when a formula is referred by many different formulas in different worksheets, which implies that one change results in the need of making a lot of little changes

16 – Switch Statements: It is accepted that complex formulas may be the cause of errors. This smell signals cell using the formulas `IF`, `LOOKUP`, `CHOOSE`, `INDEX`, `MATCH`, and `OFFSET`.

17 – Temporary Field: This smell points to hidden formulas or zero-width formula arguments, as these may be difficult to check for correctness.

4 Spectrum-based Fault Localization

In this section we describe the Spectrum-based Fault Localization (SFL) approach to software debugging, and present its application to find faults in spreadsheets.

4.1 Software Debugging with SFL

SFL is a debugging technique that calculates the likelihood of a software component being faulty [19]. SFL exploits coverage data collected from passed/failed system runs. A passed run is a program execution that is completed correctly (thus behaving as expected), and a failed run is an execution where an error was detected [11]. The criteria for determining the execution outcome can be from a variety of different sources, namely test case results and program assertions, among others. The coverage data is collected at runtime, via instrumentation, and is used to build a hit-spectra matrix.

The hit spectra of N executions constitutes a binary $N \times M$ matrix A, where M corresponds to the instrumented components of the program. In this binary matrix, each column i represents a system component and each row j represents an execution (*e.g.*, a test case). A matrix entry a_{ij} represents whether component i was involved (1) or not (0) in a particular execution j. The information of passed and failed runs is gathered in an N-length vector e, called the error vector. The pair (A, e) serves as input to the SFL technique.

After gathering the input information, the next step consists in determining what columns of the matrix A resemble the error vector e the most. This is done by quantifying the resemblance between these two vectors by means of *similarity coefficients* [20]. These coefficients are used to estimate the suspiciousness of a given software component being faulty, as its similarity coefficient (relative to the error vector) and its failure probability are directly related [21].

Several similarity coefficients do exist [11]. One of the best performing similarity coefficients for fault localization is the Ochiai coefficient [21, 22]. This coefficient was initially used in the molecular biology domain [23], and is defined as follows:

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \cdot (n_{11}(j) + n_{10}(j))}} \tag{1}$$

where $n_{pq}(j)$ is the number of runs in which the component j has been touched during execution ($p = 1$) or not touched during execution ($p = 0$), and where the runs failed ($q = 1$) or passed ($q = 0$), as depicted in the dichotomy matrix seen in Table 1. For instance, $n_{11}(j)$ counts the number of times component j has been involved in failed executions, whereas $n_{10}(j)$ counts the number of times component j has been involved in passed executions.

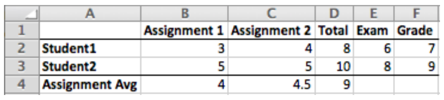
Table 1: Dichotomy Matrix of a component.

	Not Involved	**Involved**
Correct Outcome	n_{00}	n_{10}
Incorrect Outcome	n_{01}	n_{11}

The similarity coefficients that are computed can rank the system components according to their suspiciousness of containing the fault. A list of components, sorted by their similarity coefficient, is then presented to the user, helping prioritize his/her inspection of software components to pinpoint the root cause of the observed failure.

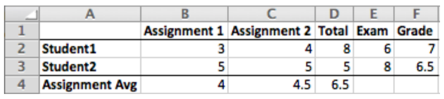
4.2 Spreadsheet Fault Localization with SFL

To better illustrate the application of SFL on the spreadsheet domain, we present a faulty spreadsheet in Figure 2, to act as a running example throughout this section. The example is adapted from [24].



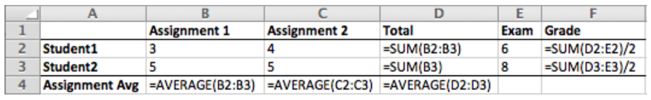
	A	B	C	D	E	F
1		Assignment 1	Assignment 2	Total	Exam	Grade
2	Student1	3	4	8	6	7
3	Student2	5	5	10	8	9
4	Assignment Avg	4	4.5	9		

(a) Correct version.



	A	B	C	D	E	F
1		Assignment 1	Assignment 2	Total	Exam	Grade
2	Student1	3	4	8	6	7
3	Student2	5	5	5	8	6.5
4	Assignment Avg	4	4.5	6.5		

(b) Faulty version.



	A	B	C	D	E	F
1		Assignment 1	Assignment 2	Total	Exam	Grade
2	Student1	3	4	=SUM(B2:B3)	6	=SUM(D2:E2)/2
3	Student2	5	5	=SUM(B3)	8	=SUM(D3:E3)/2
4	Assignment Avg	=AVERAGE(B2:B3)	=AVERAGE(C2:C3)	=AVERAGE(D2:D3)		

(c) Faulty version formula view.

Figure 2: Running example of a faulty spreadsheet.

This spreadsheet is used to calculate the grades of students enrolled in a course. It sums each student's assignments (cells `D2` and `D3`) and averages them with their exams, to compute the final grade (cells `F2` and `F3`). Average assignment grades for all students are also computed (cells `B4`, `C4` and `D4`). Figure 2a depicts the correct version of the spreadsheet and Figure 2b a faulty version of the same document. Figure 2c shows the formula view of the faulty spreadsheet. In the faulty version, the computation of the total assignment grade for Student 2 (cell `D3`) is incorrect because the cell's `SUM` formula has the wrong area as argument (i.e., `SUM(C2)` instead of `SUM(C2:C3)`). That also influences the value of Student 2's final grade (cell `F3`) and the average assignment grade (cell `D4`). This can happen, for instance, when a teacher inserts a new assignment in the grades spreadsheet for her course, and forgets to update the calculations that are influenced by that insertion. Next, we show how SFL can be used to pinpoint the fault in cell `D3`.

In order to use a traditional software debugging technique (like SFL) to aid spreadsheet fault localization, adaptations to this scope need to be performed [6, 24]. This happens because, in the spreadsheet paradigm, the concept of test case executions is non-existent. Code coverage also does not exist since there are no explicit lines of code like in traditional programming paradigms.

An *output cell* is a particular cell that is not referenced by any other cell. The set of output cells of a spreadsheet generally corresponds to the relevant information users look for in a spreadsheet as they typically are the final result of one or more computations on the data. Relating to SFL, each output cell is equivalent to a test on the system, therefore the error vector represents the correctness of the set of output cells. If we consider the running example in Figure 2, the set of output cells is

$$\textit{Output cells} = \{\texttt{F2}, \texttt{F3}, \texttt{D4}, \texttt{C4}, \texttt{B4}\}$$

To populate each row of the hit-spectra matrix, we need the coverage of each *output cell*. Although there is no concept of coverage in spreadsheets, the transitive closure of a cell's references (also known as the *cone* [6]) shares many similarities. A *cone*, which represents the data dependencies of each cell, and is given by:

$$\textit{Cone}(c) = c \cup \bigcup_{c' \in \textit{refs}(c)} \textit{Cone}(c') \tag{2}$$

where *refs*(c) is the set of cells that cell c references. A *cone* can be computed for every cell in a spreadsheet.

Table 2: Running example's output cells and their cones.

Output cell	**Cone**
`B4`	{`B2`, `B3`, `B4`}
`C4`	{`C2`, `C3`, `C4`}
`D4`	{`B2`, `B3`, `C2`, `C3`, `D2`, `D3`, `D4`}
`F2`	{`B2`, `C2`, `D2`, `E2`, `F2`}
`F3`	{`B3`, `C3`, `D3`, `E3`, `F3`}

Table 2 shows the *cones* for every *output cell* from the Figure 2 example.

The hit-spectra matrix and the error vector allow the use of the SFL algorithm to compute the failure suspiciousness of each spreadsheet cell. We have chosen the Ochiai coefficient as the suspiciousness metric for spreadsheet subjects because, according to a recent empirical study comparing similarity measures to diagnose spreadsheets [24], this coefficient was shown to be one of the best performing. Table 3 depicts the hit-spectra that feeds the SFL approach, the dichotomy matrix computations for each cell, and the result of the Ochiai similarity coefficient. Note that cell `D3`, which is highlighted with a gray background, is the faulty cell. As we can see, the faulty cell is at the top of the ranking of fault candidates.

Table 3: Hit spectra

	Spectra					Dichotomy Matrix				SFL Results	
Cell	**B4**	**C4**	**D4**	**F2**	**F3**	n_{00}	n_{01}	n_{10}	n_{11}	**Ochiai**	**Ranking**
B2	●		●	●		1	1	2	1	0.41	8
B3	●		●		●	2	0	2	1	0.58	7
B4	●					2	2	1	0	0.00	9
C2		●	●	●		1	1	2	1	0.41	8
C3		●	●		●	2	0	1	2	0.82	2
C4		●				2	2	1	0	0.00	9
D2			●	●		3	1	0	1	0.71	3
D3			●		●	3	0	0	2	1.00	1
D4			●			3	1	0	1	0.71	3
E2				●		2	2	1	0	0.00	9
E3					●	3	1	0	1	0.71	3
F2				●		2	2	1	0	0.00	9
F3					●	3	1	0	1	0.71	3
Error vector			●		●						

4.3 Misclassification of errors

As discussed in the previous subsection, the SFL approach needs a spreadsheet's output cells to be labeled as correct/incorrect to perform the diagnosis. Depending on the error detection technique used, some data can be potentially mislabeled. Therefore, high tolerance to eventual mistakes is paramount for SFL to be applicable in most real-world scenarios.

To assess how our approach behaves when the error vector is incorrectly labeled, we consider the concepts of error misclassification rate and quality of diagnosis. Error misclassification rate is the probability that an output cell is labeled as correct when it was, in fact, incorrect (or vice versa). A misclassification rate of 0.0 means that every output cell is accurately labeled. Conversely, a misclassification rate of 1.0 means that every output cell is labeled erroneously.

Quality of diagnosis is given by the following formula

$$\mathit{Quality} = 1 - \frac{|\{j|j > f, \forall_j \in S\}| + \frac{1}{2}(|\{j|j = f, \forall_j \in S\}| - 1)}{|S|} \tag{3}$$

where S is the list of the similarity coefficient values for every component and f is the similarity coefficient value of the faulty component. A quality of 1.0 means that the user only needed to inspect one component to reach the fault (i.e., the fault was at the top of the ranking). A quality of 0.0 means that, by following the ranking, the user needs to inspect all components to reach the fault.

Figure 3 shows the impact of the misclassification rate on the quality of diagnosis for different hit-spectra gathered from software programs such as Rhino[5]

[5]Available at `https://developer.mozilla.org/en-US/docs/Rhino`

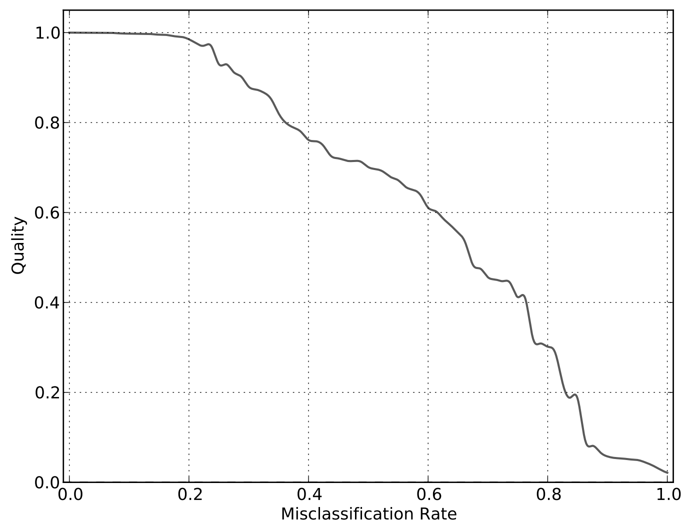


Figure 3: Quality of diagnosis versus misclassification rate.

and JHotDraw[6]. As we can see, even having 20% of error misclassification does not produce a decrease in diagnostic quality using the SFL technique. Even at 50% misclassification (meaning, in the case of the spreadsheet domain, that every other output cell is erroneously classified as correct/incorrect), the quality of diagnosis is still 0.7. SFL is, then, very resilient to mistakes in the data it consumes do compute the diagnostic, making it the ideal diagnostic algorithm to be used in conjunction with different error detection sources, such as using code smells.

This result is corroborated in the work of Hofer et al. [24, 25], which has also shown that we can consider only a subset of cones (i.e., a only a subset of the spectra's rows) to arrive at the same diagnostic accuracy.

5 The FaultySheet Detective Framework

We have extended our *SmellSheet Detective* tool [15] in order to fully implement all the smells in the spreadsheet catalog, and to implement the algorithms for fault localization. This is an extension from previous works [9], and the upgraded version of this tool was used to perform the experiments presented in this paper. This new tool is termed *FaultySheet Detective* [26].

The *FaultySheet Detective* supports both spreadsheets written in the desktop spreadsheet system Excel and spreadsheets hosted on the Google Drive cloud platform. The support for online spreadsheets was added because migration from desktop to online-based applications is becoming very common, with popular office suites seeing online versions.

[6]Available at `http://www.jhotdraw.org/`

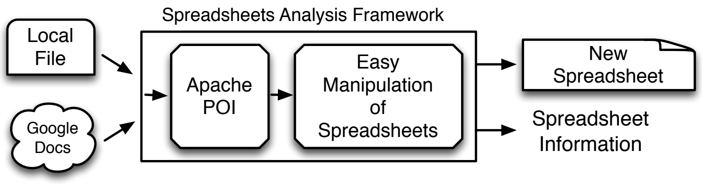


Figure 4: The Spreadsheet Analysis Framework we used.

The use of Google Drive's variant of our tool requires a registered and valid login on that platform. Our tool allows the analysis of one spreadsheet at a time, but within it the user can choose either a single worksheet or the full spreadsheet.

This tool uses the Java API Apache POI[7], in its version 3.8, for manipulating various file formats based upon the Office Open XML standards (OOXML). To read and write Microsoft's Excel files, it uses the Microsoft's OLE 2 Compound Document format API.

To be able to access Google's Drive accounts, we used the Google Data API[8]. The Google Data Protocol, which we used in version 1.47.1, is a Google owned technology for reading, writing, and modifying information on the web. To develop the *FaultySheet Detective* we used the Java version of the Google Spreadsheets API, which enables the creation of applications that read and modify data in spreadsheets stored in Google Drive accounts.

The implementation of the smells strictly follows the guidelines provided by their original source and are individual components of the tool, i.e., the tool always runs the smells individually and sequentially but filters, intercepts and processes their results as a whole. These results are used by SFL.

Figure 4 briefly describes our framework for spreadsheet analysis. Files coming either from local or network storage are used to instantiate and populate an internal Apache POI object. This object is used, via our abstractions on Apache POI, to perform a set of analysis that go from smells detection to fault localization and toxic cells analysis.

The smells are implemented using this simpler abstraction over Apache Poi and information can be displayed either as meta-information, via standard output streams or via the creation of new spreadsheets, where the user also has options to color different cells or add side information to either individual cells or the whole sheets.

The *FaultySheet Detective* together with a video demonstrating its use and the analysis framework are available at `http://ssaapp.di.uminho.pt`.

[7]`http://poi.apache.org`

[8]`https://developers.google.com/gdata/`

6 Filtering out Perceived Smells

The first step of the proposed technique consists in analyzing the individual performance in terms of error detection for all the smells already proposed in the literature. Our aim here was to filter out the smells that by their nature do not contribute to identifying spreadsheet errors, and, for all other smells, to rank their potential ability to detect errors.

In order to realize this step, we have devised the following experiment.

6.1 Experimental Setting

We analyzed a set of well-known spreadsheets – the Hawaii Kooker Corpus – containing 73 spreadsheets created from a real world problem by third-year undergraduate students and MBA students at the University of Hawaii. The errors in these spreadsheets were not "seeded" as they were naturally created by students. Part of the corpus was used in a study by Aurigemma and Panko to compare detection rates for static inspection and human inspection [13].

This corpus was developed aiming at analyzing both how end users structure their data in a spreadsheet, and how correct their solution is. This corpus also includes a correct spreadsheet to compare all others against. In order to prepare the corpus for automatically locating end-user faults with our tool, we needed to:

- First, we manually inspected and compared the correct solution to the end-users solutions. As a result of this comparison, we marked all *errors* in the spreadsheets. By *error*, we mean a cell that does not have a correct value. We also consider that a cell defined by a correct formula is an error when it depends on a cell marked with an error.

- Second, we manually defined the spreadsheets' total number of cells. The total number of cells is given by the number of cells of the smallest rectangle that contains all the non-empty cells of the spreadsheet. This consideration will not influence the computation of the `Empty Cell` smell, since in its definition an empty cell must be surrounded by non-empty ones.

6.2 Analyzing the Smells

We have performed a first analysis of the corpus without any aid from the SFL. In this step we only applied all the smells, individually, on all the spreadsheets of the corpus and gathered the results, which can be seen in Table 4.

In this table we can see a list of smells (their number and name), how many cells they pointed that really represented faulty cells (True Positives) and how many cells they marked that were perfectly correct (False Positives). The smells in this table are sorted by the descending number of true positives.

In this phase we were not concerned with sorting the smells based on how good they perform. The objective was to analyze if their correlation with actual faults. As we can see from the table, the last 6 smells cannot detect any cell

Table 4: Bad smells individual performance, ordered by the amount of true positives.

#	Smell	True Positives	False Positives
17	Temporary Field	450	564
8	Multiple References	436	114
10	Long Calculation Chain	313	29
1	Standard Deviation	305	172
7	Multiple Operations	88	13
5	Reference to Empty Cells	86	19
3	Pattern Finder	46	125
11	Duplicated Formulas	16	29
14	Middle Man	6	16
12	Inappropriate Intimacy	6	16
4	String Distance	0	1
15	Shotgun Surgery	0	0
13	Feature Envy	0	0
2	Empty Cell	0	21
6	Quasi-Functional Dependencies	0	0
9	Conditional Complexity	0	0
16	Switch Statements	0	0

with faults, with the `15` and `4` detecting only cells that contain no faults at all (false positives). From these results, we have split the smells into two groups:

Fault-Inducing Smells This group contains the smells `8`, `10`, `1`, `7`, `5`, `3`, `11`, `14`, `12`, `17`. These smells have different degrees of success, but to some extent they are all capable of detecting cells that really contain faults. This is important as we will use a two step strategy - smell detection and SFL - where the final responsibility for the results is on the latter step.

Perceived Smells This group contains the smells `4`, `15`, `13`, `2`, `6`, `9`, `16`. None of these smells was capable of detecting at least one cell with a fault. In fact, two of these smells pointed to correct cells. This result means we can discard these smells. In fact, if we used these cells to feed into our fault localization framework, we would obtain a misguided diagnosis. This happens because there was no error in the indicted input cells.

Before this study, none of these smells was ever used to detect faults, with the exception of the duplicated formulas, that was shown to be correlated with the existence of faults in spreadsheets [8].

This step fulfilled two objectives. First, we were capable of completely discarding a group of smells from the existing literature. This does not mean they are not relevant, as they still point bad practices when designing spreadsheets. It just means that, to what fault localization is concerned, they do not produce any interesting results. Second, it will provide us with a set of smells that we can use to apply our technique. In the next section, we will describe how to do so.

7 Fault-Inducing Smells Meet Spectrum-based Fault Localization

In the previous section we have selected the smells that can detect cells that have faults.

In this section we will combine this fault detection method with an SFL algorithm. This will allow to validate the faults detected by the smells, and further locate other existing faults in the spreadsheet that were not yet detected.

Next we present the algorithm we devised to implement this new technique.

7.1 The Algorithm

The algorithm of the approach is depicted in Algorithm 1. The inputs of our approach are 1) the spreadsheet under test ($\mathcal{S}$) and 2) the threshold value for the suspiciousness score given to each cell ($\mathcal{C}$).

As SFL computes a ranking of cells, sorted by their suspiciousness of containing a fault, the last input of our algorithm, $\mathcal{C}$, is used as a way to ensure that not every cell with nonzero suspiciousness gets inspected. In the software debugging domain, we often use a metric called C_e that evaluates the effort required by the user to pinpoint faulty locations. This metric indicates the number of components (for instance, statements) that the user must inspect until the fault is reached. In the spreadsheet domain, we adopted a similar route, by setting a threshold on the cells to be inspected. This threshold can be either by value (that is, only consider cells whose suspiciousness is greater than the threshold), or by percentile (that is, only consider cells whose suspiciousness is above a certain percentile). This way, we are able to not only evaluate the effort required by the user in his inspection of the diagnostic ranking, but also measure the amount of faults found and the amount of false positives given by our approach.

First, on line 1 the list of smelled cells is computed. After that, on lines 2 to 7 the cones for every cell are calculated. A cone represents the dependencies of a cell – either direct or indirect references. Also computed is the set of all cells that are references of other cells (useful for finding out output cells). The worst case time complexity of this step is $O(N^2)$, whereas the spatial complexity is $O(N)$.

On line 8, the set of output cells is calculated. Output cells are not referenced by any other cell, therefore they can be computed by subtracting the set of referenced cells to the spreadsheet. With the information about output cells, cell cones, and smelled cells, we are able to compute the inputs to SFL – a hit spectra matrix, and an error vector. As depicted in lines 10 to 17, each line of the hit spectra matrix is the cone of an output cell, and its corresponding error

Algorithm 1 Smells as input to SFL.

Input:

Spreadsheet $\mathcal{S}$

Suspiciousness Threshold $\mathcal{C}$

Output:

Diagnostic Report $\mathcal{R}$

```
1:  $\mathcal{L} \leftarrow \textsc{CalculateSmells}(\mathcal{S}, \mathcal{T})$          ▷ Compute the smell listing
2:  $\textit{references} \leftarrow \emptyset$                                              ▷ Cell references computation
3:  $\textit{cones} \leftarrow \emptyset$
4:  for all cells $c$ in $\mathcal{S}$ do
5:      $\textit{cones}[c] \leftarrow \textsc{Cone}(c)$
6:      $\textit{references} \leftarrow \textit{references} \cup \textit{cones}[c]$
7:  end for
8:  $\textit{output} \leftarrow \mathcal{S} \setminus \textit{references}$                   ▷ Output cells
9:  $M \leftarrow |\textit{output}|$
10: $\forall_{i \in \{1 \ldots M\}} : A(i) \leftarrow \emptyset$                            ▷ Hit spectra computation
11: $\forall_{i \in \{1 \ldots M\}} : e(i) \leftarrow 0$
12: for $i = 1 \rightarrow M$ do
13:     $A(i) \leftarrow \textit{cones}[\textit{output}[i]]$
14:     if $\textsc{HasSmell}(\mathcal{L}, \textit{output}[i])$ then
15:         $e(i) \leftarrow 1$
16:     end if
17: end for
18: $\mathcal{R} \leftarrow \mathrm{SFL}(A, e)$                                             ▷ Fault Localization
19: $\mathcal{R} \leftarrow \textsc{Filter}(\mathcal{R}, \mathcal{C})$                        ▷ Prune suspiciousness listing
20: $\mathcal{R} \leftarrow \mathcal{R} \cup \textsc{ToxicCells}(\mathcal{L})$
21: return $\mathcal{R}$
```

vector entry is either 1 if the output cell has a smell, and 0 otherwise. This step has a time complexity of $O(N)$ and a spatial complexity of $O(N)$.

With the hit spectra matrix A and the error vector e, the fault localization is performed, by calling the SFL method on line 18, having time complexity of $O(N^2)$ and a spatial complexity of $O(N)$.

Finally, the suspiciousness filtering step removes any component from list $\mathcal{R}$ that is below the $\mathcal{C}$ threshold. This step has both time and space complexities of $O(N)$, and the last step expands the listing to include *toxic cells*, that are cells whose references are smelly cells. These steps have a worst case space complexity of $O(N^2)$ and a time complexity of $O(N)$, where N is the total number of non-empty cells of a spreadsheet.

Overall, our approach has a worst-case time complexity of $O(N^2)$ and a spatial time complexity of $O(N)$, where N is the total number of non-empty cells of a spreadsheet.

7.2 Analyzing the Algorithm Results

The SFL algorithm uses smell-marked cells to compute a ranking of faulty cells: cells with higher rank have a higher likelihood of containing errors. We can configure our algorithm with a threshold that defines the effort to pinpoint faulty locations. Therefore, we will consider two possibilities: first, we only mark as faulty those cells whose suspiciousness score is 1.0, so that we maximize true positives. Second, we consider a percentile: we mark as faulty the top 10% of cells with highest suspiciousness.

We start by considering the first threshold. This is the approach a programmer would follow when considering the results produced by an SFL algorithm for regular programming languages: programmers will focus on searching for errors in the parts marked by the algorithm as being tagged as very suspicious.

We also want to understand if combining smells improves the results. Thus, we consider all possible combinations of 1, 2, 3, 4, and 5 smells. Figure 5 shows, on average, the percentage of cells marked by our algorithm that are in fact documented errors in the spreadsheets (i.e., the true positives). We do not consider combinations of 6 smells of more as we did not find, in the corpus we are using, a cell containing such a value: 5 was the absolute maximum.

The best results are achieved when we give as input to SFL cells detected by a single smell: on average in three cells marked as faulty, two cells do have an error, value that decreases when more smells are added. When combining 5 smells, only one cell out of two marked cells on average is an error. Intuitively, this happens because the more smells we consider, the more cells will be marked by the algorithm, and thus the more cells SFL will also consider faulty. Since the number of faults is always the same, the ratio will become worse every time a new smell is considered.

Next, we present the results of combining each of the 10 *fault-inducing smells* individually with SFL. Figure 6 shows, for each smell, the ratio of true positives yielded by the technique over its false positives (left y-axis), and the percentage of errors found (right y-axis).

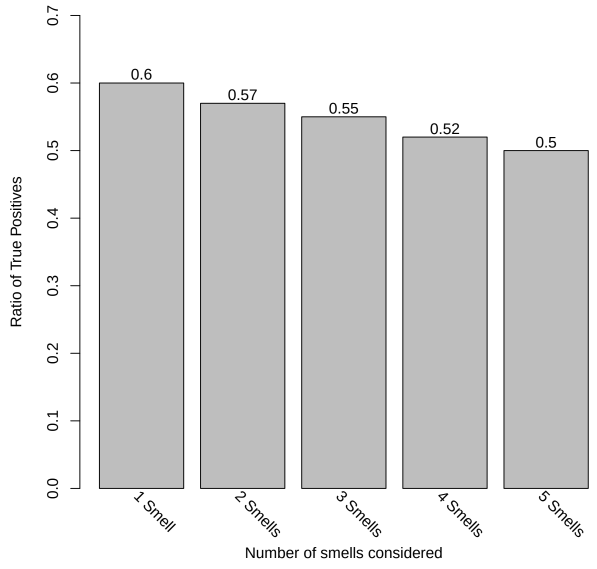


Figure 5: Percentage of true positives.

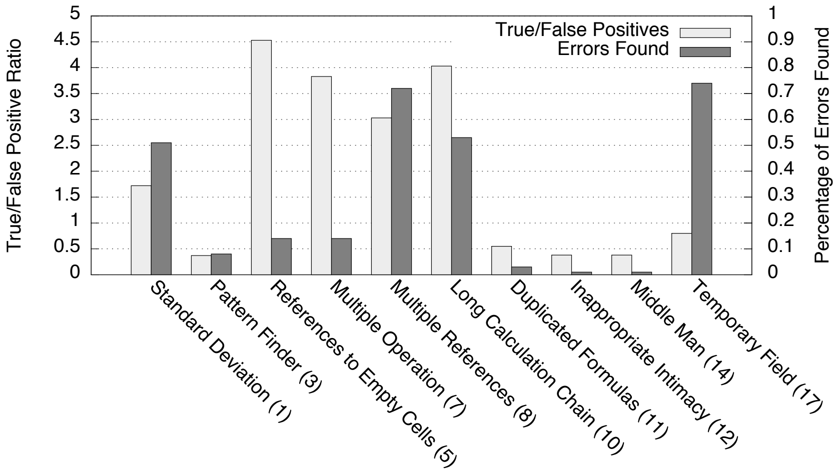


Figure 6: True positive over false positives and percentage of errors found using single smells.

We can see that the five smells that combined with SFL locate more errors are: *Temporary Field* with 74.0% of the total errors,*Multiple References* with 72.5,%*Long Calculation Chain* with 53.1%, *Standard Deviation* with 50.6% and*Multiple Operations* with 14.5% of the total errors found. These are also the smells that produce the best true/false positives ratio. For example, *Multiple References* produces 3 times more true positives than false ones. On practice this means that every three out of four marked cells contain an error.

In Figure 5 we can see the average results of combining different smells. Next, we consider the combination of the best smells, according to the results shown in Figure 6. The results of the combinations of 1 up to 5 smells are presented in Figure 7. Again, the left y-axis represents the ratio of real faults over all the marked (smelly) cells, and the right y-axis represents the real smells over the real faults.

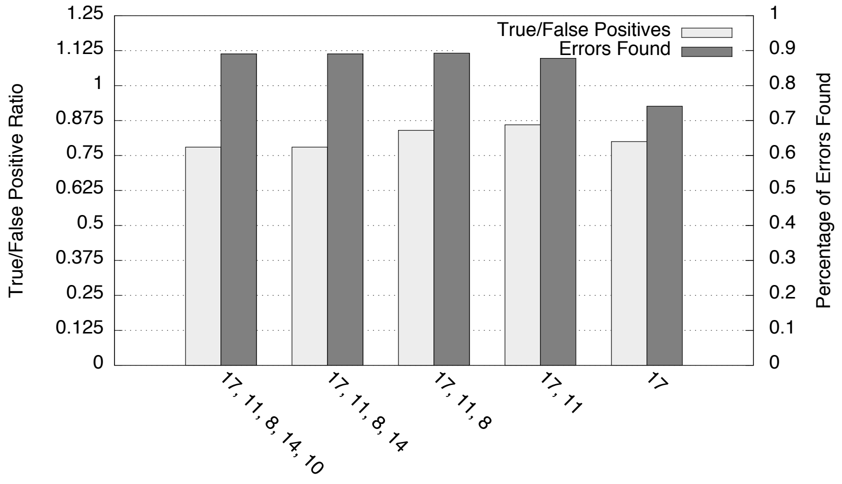


Figure 7: Combination of 1 up to 5 of the smells with better results.

As expected, by giving more smelly cells to the SFL algorithm the number of detected errors increases. We are able to locate 89.1% of errors with this setting. However, because different smells may mark the same cells as smelly (this is the case of smells 8 and 10), this represents only a small improvement with a high cost: the number of false positives doubles, which increases the work of a user of our technique to detect errors.

Next, let us consider the 10% percentile threshold, which marks as faults 10% of the marked cells with highest suspiciousness. Figure 8 presents such results when considering the combination of (up to) five smells.

It is quite clear that the best case scenario in this case is worst than the worst case scenario in the approach where we use only the faults marked by SFL with the 100% threshold (see Figure 5).

We can conclude that the combination of smells and SFL produces good results when a single smell is considered and a suspiciousness score threshold of 1.0. A combination of smells does (slightly) improve the number of detected

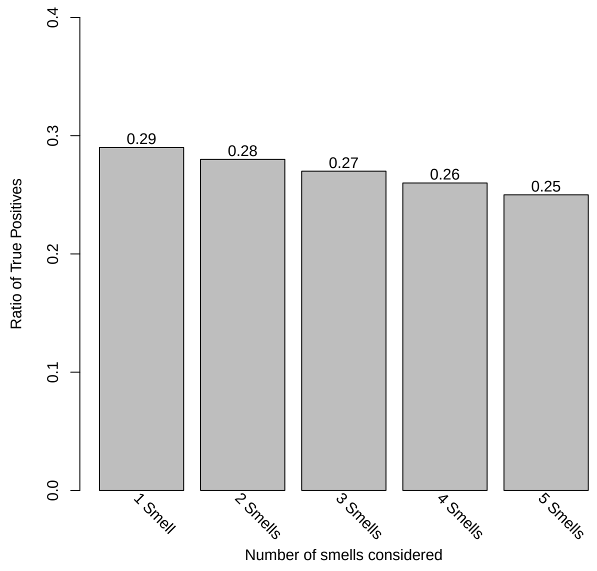


Figure 8: Percentage of cells that contain errors from SFL's top 10% results.

errors, but at a high cost: the increase of false positives, which implies an increase in the work to locate faults in a spreadsheet.

7.3 Threats to validity

The main threat to external validity of these empirical results is the fact that, although the subjects were all real spreadsheets, it is plausible to assume that a different set of subjects, having inherently different characteristics, may yield different results. This is particularly true if we consider inter-worksheet smells.

A second threat is the fact that we use only one corpus of spreadsheets. Although this is the case, it is also true that it is the largest corpus of spreadsheets with documented errors, and it was created with the goal of providing a research corpus for spreadsheet error detection. This is a necessary condition to execute such a study. If the errors are not known, we can not provide an analysis that is not speculative.

Threats to internal validity are related to faults in our underlying implementation, such as smell detection, hit spectra generation, or fault localization. To minimize this risk, some testing and individual result checking were performed before the experimental phase.

8 FaultySheet Detective Meets Correct Spreadsheets

In the previous analysis we have classified a set of 17 smells, and used the ones that can actually point to faulty cells together with a SFL algorithm to evaluate their efficacy under a corpus of spreadsheets with faults. These results were obtained using a corpus of spreadsheets that we know contain errors, which further more are documented.

The setting used in the previous analysis provided a good subject to perform our analysis because the documented errors avoid threats to validity that would appear if we were to manually insert errors in these sheets. We could be biased towards introducing errors that we know would be watched by our tool.

There are available corpus of spreadsheets, such as the one presented in [27], that is hard to use in our setting because it is likely errors exist but are not documented. This means we would have to necessarily locate all the errors before an analysis is possible, which an extremely hard task that would create obvious threats to validity.

In this section we will present the findings of running our tool in a corpus of spreadsheets that are known not to contain faults. We are motivated towards using the corpus exactly as it is, as inserting errors would threat our results for the reasons presented in the previous paragraph. Therefore, we shifted our analysis towards a different evaluation of our tool.

In an ideal case, a tool that detects errors would return no results on a setting where no errors exist, so we tested the performance of our technique in a setting where ideally no cells should be marked. This new analysis will describe the behavior of our technique regarding the number of false positives pointed out on an environment with our errors.

With this new evaluation, we intend to show that our technique is well behaved when no faults are present at the spreadsheets under analysis.

The spreadsheets we are using are the ones available through a CD with the book "The Art of Modeling with Spreadsheets" [28]. Although it may be the case that these spreadsheets contain errors, this scenario is highly unlikely, as this is a well-knows book on modeling spreadsheets, currently on the 4th edition.

This set consists of 27 spreadsheets, each containing between 1 and 16 worksheets, with a total of 121 worksheets, and a total of 593084 cells. More than half of the worksheets, 66 out of the 121, contain formulas. The spreadsheets represent different realms that range from spreadsheets to make the decision of how much to bid for the salvage rights to a grounded ship, modeling the production at Delta Oil, advertising budget, or planning shipments.

Some pre-processing was necessary for these spreadsheets before performing our analysis. For example, some cells from the spreadsheet `Hastings1` were deleted because they were calculated using macros, which we cannot work with yet. Another talk was to remove in the spreadsheets the array formulas used as our tool cannot parse them (ex., for `AdBudget`, where this accounts only for 8 cells). The spreadsheets `Hastings2` and `Options` were removed from the analysis because these spreadsheets were heavily based on macros, and removing the cells that use them makes the spreadsheets completely irrelevant.

8.1 Results

After running our tool through all the spreadsheets in the corpus a total of 9122 cells were marked. As explained above, this corpus contains no errors so we immediately know these are false positives. Comparing these to the total number of cells of all the spreadsheets on this corpus (593084), this gives a total of 1.54% of cells wrongly marked by our algorithm as potentially being faulty. While this results is not perfect (which would be 0%) it proves our technique is well behaved when analyzing correct spreadsheets.

It is important to recall that our technique is based on the detection of bad smells, which may not necessarily represent faults. For this reason, it would be very unlikely that our tool would not mark any cell, as it would mean more than a corpus of correct spreadsheets, it would mean we are dealing with a corpus of spreadsheets which are perfectly designed.

This results reveal that the spreadsheets used in [28] are in fact very good examples of well designed spreadsheets, as only less then 2% of the cells were marked as being smelly. This shows that these spreadsheets are indeed much better, from a design perspective, than other available corpus [8].

8.2 Detailed Results

Although the overall results are very good, they deserve further analysis. In Figure 9 we present the detailed findings for each of the spreadsheets in the corpus.

The best performance is achieved in the `Dish` spreadsheet, where only 0.55% (372 out of 67246) of the cells are marked by our tool. A similar resulted is achieve in the `Analgesics` spreadsheet (1466 out of 195488) where only 0.75% of all the cells are marked.

One conclusion from the analysis of this chart is that the biggest spreadsheets are the ones with best results. The reason this happens is because these spreadsheets are more data oriented, which means they contain less formulas and more informations. The good results are therefore obtained because a) most of our smells actually work on formulas and b) these spreadsheets are huge, so it is easier for the percentage of cells marked to be slower.

Nevertheless, for some small spreadsheets we can also achieve good results. For instance, for `SS Kuniang` we mark only 2.2% (7 out of 315) cells, and for `Hastings1` we mark only 4.5% (10 out of 224) of the cells.

On the other hand the worst scenario occurs for the spreadsheet `Diffusion`, where 44.4% (128 out of 288) of all cells are pointed by the tool. The second worst is obtained on the spreadsheet `Forecasting` where 24.2% (443 out of 1833) of the cells are marked. It is interesting no notice there is no relation between the worst cases and the size of the spreadsheets.

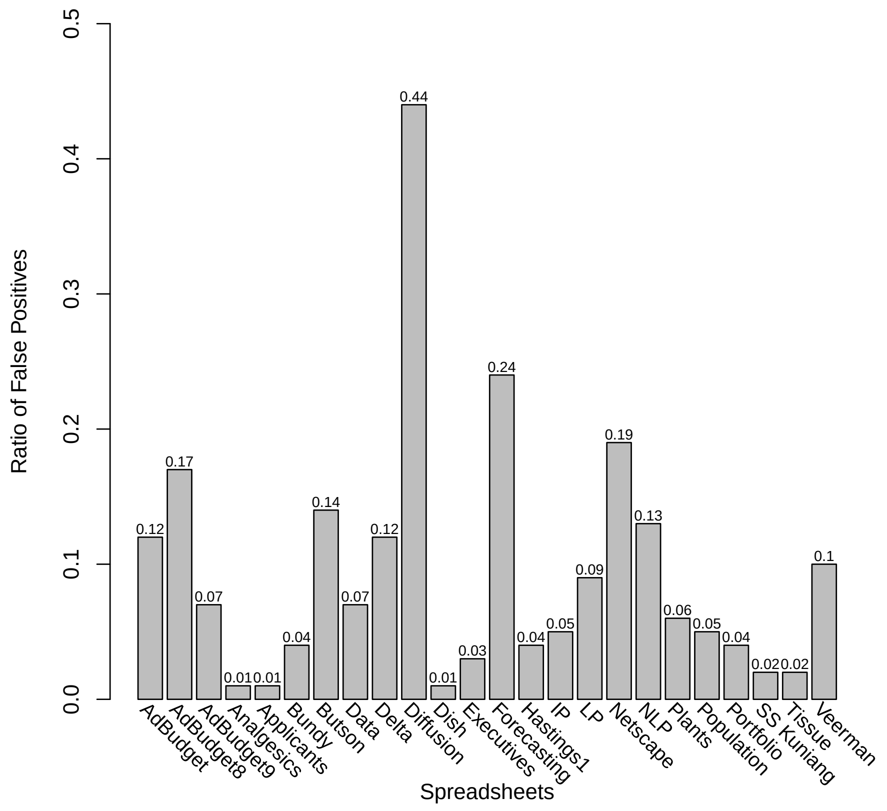


Figure 9: False positive ratio for each spreadsheet.

9 Related Work

The work presented in this paper focuses on identifying smells in a spreadsheet and feeding them into a fault localization framework. Other efforts related to using spreadsheet smells and metrics to detect errors in spreadsheets have been proposed before [29, 30]. Hermans *et al.* [7] proposed an approach to locate spreadsheet smells and communicate them to users via data flow diagrams. Recently, an approach to detect and visualize data clones (i.e., formulas whose values are copied as plain text to a different location) was also described [31].

AmCheck [32] also uses smell detection in its analysis. This tool detects a particular kind of smells, coined ambiguous computation smells, which frequently appear in cells that do not maintain the computational semantics of their row or column. AmCheck also provides repair suggestions by inferring formula patterns for the smelly cells that maintain the semantics of their surroundings.

GoalDebug [33] is a spreadsheet debugger targeted at end users. Whenever the computed output of a cell is incorrect, the user can supply that cell's expected value. This expected value is used by the system to generate a list of change suggestions for cell formulas, ranked using a set of heuristics. A drawback of this approach is that users are expected to detect errors in the spreadsheet, and provide the system with the correct output value. In our approach, the error detection phase is automated by testing spreadsheets against our smell catalog.

CheckCell [34] is an analysis tool to automatically find potential data errors in spreadsheets. It locates data with disproportionate impact on computations, with the assumption that, especially in data-intensive scenarios, cells that have an unusually high impact are either very important or incorrect, so they should require further inspection.

Kulesz et al. [35] propose an interactive approach to debugging via creating additional test cases that are decoupled from the spreadsheet. However, users still have to provide the new input values and the expected output value for every test they create. Another approach that leverages test cases to locate faults in spreadsheets was proposed by Hofer et al. [36]. Instead of requiring the user to provide test inputs, it leverages diagnostic and repair approaches such as model-based debugging [37] and genetic programming [38] to generate possible repair mutations. The approach would generate the inputs of a distinguishing test case where the computed values for two mutated versions of a spreadsheet would differ on the same input. As a way to filter the repair mutations, the user is then asked to select the test cases which compute the correct value.

There are several other spreadsheet analysis tools that try to find inconsistencies in spreadsheet formulas [39, 40, 41, 42, 43, 44], which differ in the rules they employ and the amount of user effort required to provide additional input. Most of these approaches require the user to annotate the spreadsheet cells with additional information. An exception is the UCheck system [45], which can perform unit analysis automatically by exploiting header inference techniques [39].

Other approaches that aim at minimizing the occurrence of errors in spreadsheets include code inspection [46], refactoring [10, 47], model-driven engineering [48, 49], and the adoption of better spreadsheet design practices [50, 51], but none of these approaches focuses on spreadsheets' debugging.

10 Conclusion

In this paper we described an approach to automatically locate faults in spreadsheets. This approach uses a catalog of 15 well-known documented spreadsheet smells to perform smell detection and provide an indication of possible faults in the spreadsheet.

This set of smells was divided into two: one containing smells that actually point out faulty smells, and another with the smells that cannot find cells with faults.

The cells detected by the first set of smells are fed into a spectrum-based fault localization framework, commonly used in the software debugging field, as a way to improve the quality of the diagnosis. Our empirical experiments, using a well-known faulty spreadsheet catalog, have shown that our approach is able to detect more than 70% of errors in spreadsheets in a setting where two out of three identified faulty cells are documented errors.

There are several research questions that still require further investigation. First, we plan to provide natural and intuitive visualizations to improve user's comprehension of diagnostic data. Second, we plan to study ways to provide fix suggestions to users, namely by mutating spreadsheets [52].

Acknowledgements

This works was partially supported by the Project NORTE-07-0124-FEDER-000062, co-financed by the North Portugal Regional Operational Programme (ON.2 - O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF).

References

[1] Panko RR, Ordway N. Sarbanes-oxley: What about all the spreadsheets? *arXiv preprint arXiv:0804.0797* 2008; .

[2] Hermans F, Pinzger M, van Deursen A. Supporting professional spreadsheet users by generating leveled dataflow diagrams. *Proceedings of the International Conference on Software Engineering (ICSE'11)*, ACM, 2011; 451–460.

[3] Panko RR. Spreadsheet errors: What we know. what we think we can do. *arXiv preprint arXiv:0802.3457* 2008; .

[4] Panko R. Facing the problem of spreadsheet errors. *Decision Line, 37(5)* 2006; .

[5] Reinhart CM, Rogoff KS. Growth in a time of debt. *American Economic Review* September 2010; **100**(2):573–78, doi:10.1257/aer.100.2.573.

[6] Hofer B, Riboira A, Wotawa F, Abreu R, Getzner E. On the empirical evaluation of fault localization techniques for spreadsheets. *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. Springer, 2013; 68–82.

[7] Hermans F, Pinzger M, Deursen Av. Detecting and visualizing inter-worksheet smells in spreadsheets. *Proceedings of the International Conference on Software Engineering (ICSE'12)*, IEEE Press, 2012; 441–451.

[8] Hermans F, Pinzger M, van Deursen A. Detecting code smells in spreadsheet formulas. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'12)*, IEEE, 2012; 409–418.

[9] Cunha J, Fernandes JP, Ribeiro H, Saraiva J. Towards a catalog of spreadsheet smells. *Proceedings of the International Conferences on Computational Science and Its Applications (ICCSA'12)*. Springer, 2012; 202–216.

[10] Badame S, Dig D. Refactoring meets spreadsheet formulas. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'12)*, IEEE, 2012; 399–409.

[11] Abreu R, Zoeteweij P, Golsteijn R, Van Gemund AJ. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 2009; **82**(11):1780–1792.

[12] Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Boston, MA, USA, 1999.

[13] Aurigemma S, Panko RR. The detection of human spreadsheet errors by humans versus inspection (auditing) software. *arXiv preprint arXiv:1009.2785* 2010; .

[14] Abreu R, Cunha J, Fernandes JP, Martins P, Perez A, Saraiva J. Smelling faults in spreadsheets. *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, IEEE, 2014; 111–120.

[15] Cunha J, Fernandes JP, Martins P, Mendes J, Saraiva J. Smellsheet detective: A tool for detecting bad smells in spreadsheets. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'12)*, IEEE, 2012; 243–244.

[16] Asavametha A. Detecting bad smells in spreadsheets. Master's Thesis, Oregon State University June 2012.

[17] Chiang F, Miller RJ. Discovering data quality rules. *Proceedings of the VLDB Endowment* 2008; **1**(1):1166–1177.

[18] Codd EF. A relational model of data for large shared data banks. *Communications of the ACM* 1970; **13**(6):377–387.

[19] Abreu R, Zoeteweij P, Van Gemund AJ. On the accuracy of spectrum-based fault localization. *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, IEEE, 2007; 89–98.

[20] Jain A, Dubes R. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[21] Abreu R, Zoeteweij P, Van Gemund AJ. An evaluation of similarity coefficients for software fault localization. *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, IEEE, 2006; 39–46.

[22] Lucia L, Lo D, Jiang L, Thung F, Budi A. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process* 2014; **26**(2):172–219.

[23] da Silva Meyer A, Franco Farcia AA, Pereira de Souza A. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*Zea mays* L). *Genetics and Molecular Biology* 2004; **27**(1):83–91.

[24] Hofer B, Perez A, Abreu R, Wotawa F. On the empirical evaluation of similarity coefficients for spreadsheets fault localization. *Automated Software Engineering* 2014; :1–28.

[25] Hofer B. Spectrum-based fault localization for spreadsheets: Influence of correct output cells on the fault localization quality. *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'14)*, 2014; 263–268, doi:10.1109/ISSREW.2014.100.

[26] Abreu R, Cunha J, Fernandes JP, Martins P, Perez A, Saraiva J. FaultySheet detective: When smells meet fault localization. *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, vol. 14, 2014.

[27] Hermans F, Murphy-Hill E. Enron's spreadsheets and related emails: A dataset and analysis. *Proceedings of the International Conference on Software Engineering (ICSE'15) (to appear)*. Firenze, Italy, May 2015; .

[28] Powell SG, Baker KR. *The Art of Modeling with Spreadsheets*. John Wiley & Sons, Inc.: New York, NY, USA, 2003.

[29] Bregar A. Complexity metrics for spreadsheet models. *arXiv preprint arXiv:0802.3895* 2008; .

[30] Hodnigg K, Mittermeir RT. Metrics-based spreadsheet visualization: Support for focused maintenance. *arXiv preprint arXiv:0809.3009* 2008; .

[31] Hermans F, Sedee B, Pinzger M, Deursen Av. Data clone detection and visualization in spreadsheets. *Proceedings of the International Conference on Software Engineering (ICSE'13)*, IEEE Press, 2013; 292–301.

[32] Dou W, Cheung SC, Wei J. Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation. *Proceedings of the International Conference on Software Engineering (ICSE'14)*, ACM, 2014; 848–858.

[33] Abraham R, Erwig M. Goaldebug: A spreadsheet debugger for end users. *Proceedings of the International Conference on Software Engineering (ICSE'07)*, IEEE Computer Society, 2007; 251–260.

[34] Barowy DW, Gochev D, Berger ED. Checkcell: data debugging for spreadsheets. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*, ACM, 2014; 507–523.

[35] Kulesz D, Scheurich J, Beck F. Integrating anomaly diagnosis techniques into spreadsheet environments. *In PRoceedings of the IEEE Working Conference on Software Visualization (VISSOFT'14)*, IEEE, 2014; 11–19.

[36] Hofer B, Abreu R, Perez A, Wotawa F. Generation of relevant spreadsheet repair candidates. *Proceedings of the European Conference on Artificial Intelligence (ECAI'14)*, 2014; 1027–1028.

[37] Hofer B, Wotawa F. Why does my spreadsheet compute wrong values? *Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2014; 112–121.

[38] Hofer B, Wotawa F. Mutation-based spreadsheet debugging. *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'13)*, 2013; 132–137.

[39] Abraham R, Erwig M. Header and unit inference for spreadsheets through spatial analyses. *Proceedings of the IEEE Symposium on Software Visualization (VISSOFT'04)*, IEEE, 2004; 165–172.

[40] Ahmad Y, Antoniu T, Goldwater S, Krishnamurthi S. A type system for statically detecting spreadsheet errors. *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE'03)*, IEEE, 2003; 174–183.

[41] Erwig M, Burnett M. Adding apples and oranges. *Practical Aspects of Declarative Languages.* Springer, 2002; 173–191.

[42] Abreu R, Riboira A, Wotawa F. Debugging spreadsheets: a csp-based approach. *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'13)*, IEEE, 2012; 159–164.

[43] Abreu R, Hofer B, Perez A, Wotawa F. Using constraints to diagnose faulty spreadsheets. *Software Quality Journal* 2014; :1–26.

[44] Jannach D, Baharloo A, Williamson D. Toward an integrated framework for declarative and interactive spreadsheet debugging. *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'13)*, 2013; 117–124.

[45] Abraham R, Erwig M. UCheck: A spreadsheet type checker for end users. *Journal of Visual Languages and Computing* 2007; **18**(1):71–95.

[46] Panko RR. Applying code inspection to spreadsheet testing. *Journal of Management Information Systems* 1999; :159–176.

[47] Hermans F, Dig D. Bumblebee: A refactoring environment for spreadsheet formulas. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, FSE 2014, ACM: New York, NY, USA, 2014; 747–750, doi:10.1145/2635868.2661673.

[48] Cunha J, Fernandes JP, Mendes J, Saraiva J. Embedding, evolution, and validation of spreadsheet models in spreadsheet systems. *IEEE Transactions on Software Engineering* 2014; In press.

[49] Cunha J, Fernandes JP, Martins P, Pereira R, Saraiva J. Refactoring meets model-driven spreadsheet evolution. *Proceedings of International Conference on the Quality of Information and Communications Technology (QUATIC'14)*, 2014; 196–201.

[50] Cunha J, Erwig M, Saraiva J. Automatically inferring classsheet models from spreadsheets. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'10)*, IEEE, 2010; 93–100.

[51] Cunha J, Fernandes JP, Mendes J, Saraiva J. MDSheet: A Framework for Model-driven Spreadsheet Engineering. *Proceedings of the International Conference on Software Engineering (ICSE'12)*, ICSE '12, ACM, 2012; 1412–1415.

[52] Abraham R, Erwig M. Mutation operators for spreadsheets. *IEEE Transactions on Software Engineering* 2009; **35**(1):94–108.