Smelling Faults in Spreadsheets

Rui Abreu
HASLab/INESC TEC & Universidade do Porto, Portugal
Email: rui@computer.org

Jácome Cunha
HASLab/INESC TEC & Universidade do Minho, Portugal
Email: jacome@di.uminho.pt

João Paulo Fernandes
HASLab/INESC TEC & RELEASE, Universidade da Beira Interior, Portugal
Email: jpf@di.ubi.pt

Pedro Martins
HASLab/INESC TEC & Universidade do Minho, Portugal
Email: prmartins@di.uminho.pt

Alexandre Perez
HASLab/INESC TEC & Universidade do Porto, Portugal
Email: alexandre.perez@fe.up.pt

João Saraiva
HASLab/INESC TEC & Universidade do Minho, Portugal
Email: jas@di.uminho.pt

***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, among other consequences, serious economical impacts on businesses.**

This paper proposes a technique to automatically pinpoint potential faults in spreadsheets. The technique proposed combines a catalog of spreadsheet smells that provides a first indication of an eventual 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 one well known and well documented catalog of faulty spreadsheets. Our experiments provide two main results: we have filtered smells that point faulty cells from smells that are not capable of doing so, and we provide a technique capable of detecting a significant number of errors: two thirds of the identified faulty cells are in fact (documented) errors.

I. 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 their 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. firms use spreadsheets for financial reporting [1], that 90% of all analysts in the industry perform calculations in spreadsheets [1] and that 50% of all spreadsheets are the basis for decisions in companies [2].

This importance, however, has not been achieved together 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].

One particularly sad 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 latter 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. In this line, 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 *good* bad smells, i.e., the smells that are capable, on their own, of signaling spreadsheet errors and ii) the one of *bad* bad 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 the good bad smells are provided as input to a fault localization algorithm. In this step, we seek to confirm the faulty nature of certain cells, and we also attempt to identify spreadsheet cells that may have contributed to a fault.

Finally, we have extensively evaluated the debugging method that we propose, again by using the corpora of [13]. Our results show that in 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.

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

This paper makes the following contributions:

- we have implemented in a tool a state-of-the-art and extensive catalog of spreadsheet smells;
- we have made a first distinction of these smells by their natural ability of signaling, or not, spreadsheet errors;
- we proposed a method that, for the first time, combines smell detection with spectrum-based fault localization;
- our method has also been fully implemented in a tool;
- for a concrete and independent set of documented spreadsheets, we have analyzed the results that we obtain on them by our method.

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

II. Motivation

This section motivates our approach using a spreadsheet taken from the corpus of spreadsheets we use to validate our techniques [13].

This spreadsheet can be seen in Figure 1a and represents the estimated expenses and revenues of a company, with parameters like Labor, Rent, and Taxes. This spreadsheet has ten cells whose observed values are wrong (marked in red[2]). As the net income depends on other values that are wrong, 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 to the spreadsheet, individually or in combination, and signal 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 of a fault localization algorithm to discover 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 potential faulty.

Let us consider the spreadsheet of Figure 1a. Just applying a single spreadsheet smell - *Multiple References* [8] - combined with the fault-localization algorithm allows us to identify the errors displayed 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 results. That is to say that it did not marked as faulty any correct cell. In this paper we consider a full catalog of spreadsheet smells and a large corpus of documented faulty spreadsheets and our approach is able to locate more more than 70% of all faulty cells. About 2 out of 3 cells that we identify correspond to a (documented) fault.

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

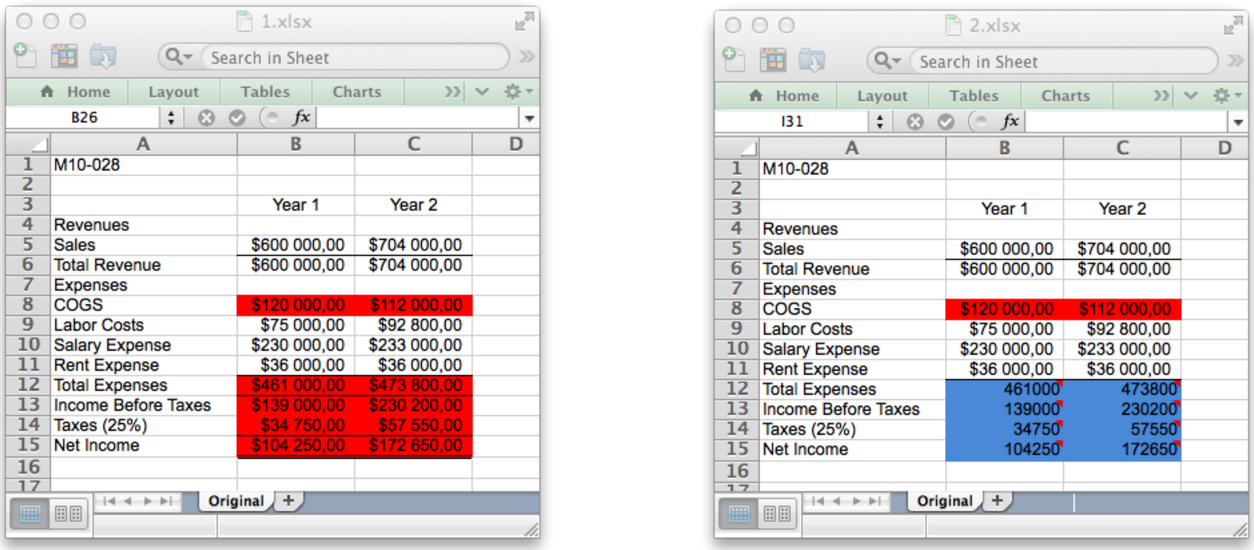
III. Spreadsheet Smells

The concept of *code smell* (or 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: a method with ten arguments, 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. In the context of our work, we have taken the union of all the proposed catalogs, obtaining the comprehensive list that we review next. The first six smells in this list were proposed in [9], [14], and exploit, for example, statistical properties of spreadsheet data in the same row/column. The following five smells have appeared in [8], and refer to spreadsheet formulas. Finally, the last four smells in this list deal with inter-worksheet smells [7]. Each smell has a number which will be used latter 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 or a 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.
- 6 - *Quasi-Functional Dependencies*: In [15] it is described a technique to identify dirty values using a slightly relaxed version of Functional Dependencies (FD) [16]. 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. This smell flags situations where equal values in a column correspond to the same values in another column, 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



	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	$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	

(a) Spreadsheet with errors.

	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	

(b) Spreadsheet with faults located.

Fig. 1: The same spreadsheet before and after our faults detection technique is applied.

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 it 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

IV. 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.

A. *Software Debugging with SFL*

SFL is a debugging technique that calculates the likelihood of a software component being faulty [17]. 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 touched (1) or not (0) during 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 for 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* [18]. 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 [19].

Several similarity coefficients do exist [11]. One of the best performing similarity coefficients for fault localization is the Ochiai coefficient [19], [20]. This coefficient was initially used in the molecular biology domain [21], and is defined as follows:

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

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 ($q = 0$). 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. Formally, $n_{pq}(j)$ is defined as:

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}| \quad (2)$$

In fact, the Ochiai coefficient can be regarded as the cosine between two vectors in n-dimensional space.

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.

B. Spreadsheet Fault Localization with SFL

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], [22]. 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.

As an alternative to code coverage, cells and cell references can be used to compute a hit-spectra matrix, one of the inputs to the SFL technique. Here, a *cone* represents the data dependencies of each cell, and is given by:

$$\mathit{Cone}(c) = c \cup \bigcup_{c' \in \mathit{refs}(c)} \mathit{Cone}(c') \quad (3)$$

where *refs*(c) is the set of cells that cell c references. For each cell, its *cone* can be computed.

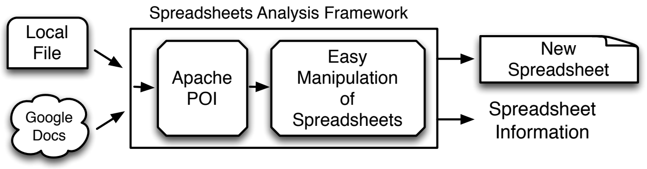


Fig. 2: The Spreadsheet Analysis Framework we used.

From these *cones*, the hit-spectra matrix can be generated, where each row j has the dependencies of the output cell c_j. Output cells of a spreadsheet are the set of cells that are not referenced by any other cell. As each row of the matrix corresponds to one of the output cells, the error vector represents their correctness.

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 [22], this coefficient was shown to be one of the best performing.

V. THE *SmellSheet Detective* FRAMEWORK

We have extended our *SmellSheet Detective* tool [14] in order to fully implement all the smells in the spreadsheet catalog, and to implement the algorithms for fault localization. This is an an extension from previous works [9], and the upgraded version of this tool was used to perform the experiments presented in this paper.

The *SmellSheet 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.

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 APIs Apache POI[3], in its version 3.8, for manipulating various file formats based upon the Office Open XML standards (OOXML) and Microsoft's OLE 2 Compound Document format (OLE2). Basically, using this API one can easily read and write Microsoft's Excel files.

To be able to access Google's Drive accounts, we used the Google Data API[4]. 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 *SmellSheet 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.

[3]http://poi.apache.org

[4]https://developers.google.com/gdata/

The implementation of the smells strictly follows the guidelines provided by their original source and are individual and exclusive 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 2 briefly describes our framework for spreadsheets 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 *SmellSheet Detective* together with a video demonstrating its use and the analysis framework are available at http://ssaapp.di.uminho.pt.

VI. Filtering out Bad Bad Smells

The first step of the new technique that we propose in this paper consisted 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 bad 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.

A. Experimental Setting

We analyzed a set of well-known spreadsheets, the ones that form 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 total number of cells of the spreadsheets. 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.

B. 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 marked the results, which can be seen in Table I.

TABLE I: Bad smells individual performance.

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

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 ordered by the descending number of true positives.

On this phase we were not concerned with sorting the smells based on how good they perform. The objective was to analyze if there were some that would provided no interesting results at all, which happened. As we can see from the table, the last 6 smells cannot detect any cell 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:

a) Good Bad Smells: This group contains the smells `11`, `8`, `14`, `10`, `6`, `12`, `3`, `9`, `7`. 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.

b) Bad Bad Smells: This group contains the smells `15`, `13`, `5`, `4`, `2`, `1`. 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. Even though we are counting on the SFL to process the results from applying the smells, there would be absolutely nothing it could do with the results of this group. Indeed, it would misguide the SFL process.

This step fulfilled two objectives. First, we were capable of completely discharging a group of smells from the existing literature. This does not mean they are not relevant, as they still point bad practices and design in spreadsheets. It just means that, to what fault localization is related, 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.

VII. Good Bad Smells Meet Spectrum-based Fault Localization

In the previous section we have selected the smells that can produce good results, that is, that can point out cells that have faults.

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

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

A. 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 ($\mathcal{C}$).

Algorithm 1 Smells as input to SFL.

```
Input:
        Spreadsheet $\mathcal{S}$
        Suspiciousness Threshold $\mathcal{C}$
Output:
        Diagnostic Report $\mathcal{R}$
1: $\mathcal{L} \leftarrow$ CALCULATESMELLS($\mathcal{S}, \mathcal{T}$)   ▷ Compute the smell listing
2: $references \leftarrow \emptyset$   ▷ Cell references computation
3: $cones \leftarrow \emptyset$
4: for all cells $c$ in $\mathcal{S}$ do
5:     $cones[c] \leftarrow$ CONE($c$)
6:     $references \leftarrow references \cup cones[c]$
7: end for
8: $output \leftarrow \mathcal{S} \setminus references$   ▷ Output cells
9: $M \leftarrow |output|$
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:     $A(i) \leftarrow cones[output[i]]$
14:     if HASSMELL($\mathcal{L}, output[i]$) then
15:         $e(i) \leftarrow 1$
16:     end if
17: end for
18: $\mathcal{R} \leftarrow$ SFL($A, e$)   ▷ Fault Localization
19: $\mathcal{R} \leftarrow$ FILTER($\mathcal{R}, \mathcal{C}$)   ▷ Prune suspiciousness listing
20: $\mathcal{R} \leftarrow \mathcal{R} \cup$ TOXICCELLS($\mathcal{L}$)
21: return $\mathcal{R}$
```

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 insure 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 (e.g., 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 (i.e., only consider cells whose suspiciousness is greater than the threshold), or by percentile (i.e., 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 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.

B. Analyzing the Algorithm Results

The SFL algorithm uses cells marked by the smells to compute a ranking of faulty cells: cells with higher rank have a higher probability 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 mark as faulty cells whose suspiciousness is 100% only, so that we maximize true positives. Second, we consider a percentile: we mark as faults 10% of the 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 3 shows, in average, the percentage of cells marked by our algorithm that are in fact documented errors in the spreadsheets. 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.

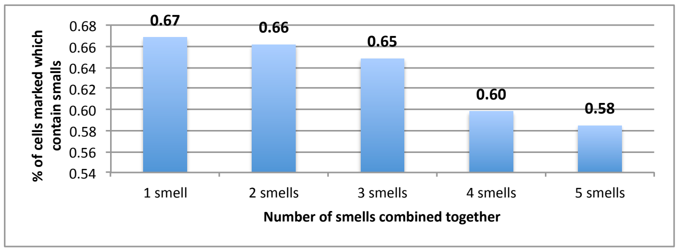


Fig. 3: Percentage of cells marked by the algorithm that are in fact errors.

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 in average is a error.

Next, we present the results of combining each of the 9 *Good Bad Smells* individually with SFL. Figure 4a shows, for each smell, the percentage of errors found over the total number of marked cells, and the percentage of errors found over the total of existing errors.

We can see that the five smells that combined with SFL locate more errors are: *Multiple References* with 72.5% of the total errors, *Long Calculation Chain* with 53.1%, *Standard Deviation* with 50.6%, *Multiple Operations* with 14.5% and *References to Empty Cells* with 14.2% of the total errors found. These are also the smells that produce the best true/false positives relation. For example, *Multiple References* produces 3 times more true positives than false ones. On practice this means that every three out of four marked cell contains an error.

In Figure 3 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 4a. The results of the combinations of 1 up to 5 smells are presented in Figure 4b.

As expected, by giving more smelly cells to the SFL algorithm the number of detected errors increases. We are able to locate 77.9% 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 5 presents such results when considering the combination of (up to) five smells.

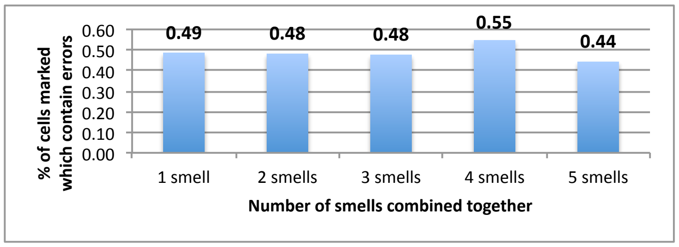


Fig. 5: Percentage of cells that contain errors using the 10% best results from SFL.

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 3).

We can conclude that the combination of smells and SFL produces very good results when a single smell is considered and a 100% suspiciousness threshold is used. A combination of smells does (slightly) improves the number of detected errors, but at a high cost: the increase of false positives which implies an increase in the work to locate faults in a spreadsheet.

C. 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.

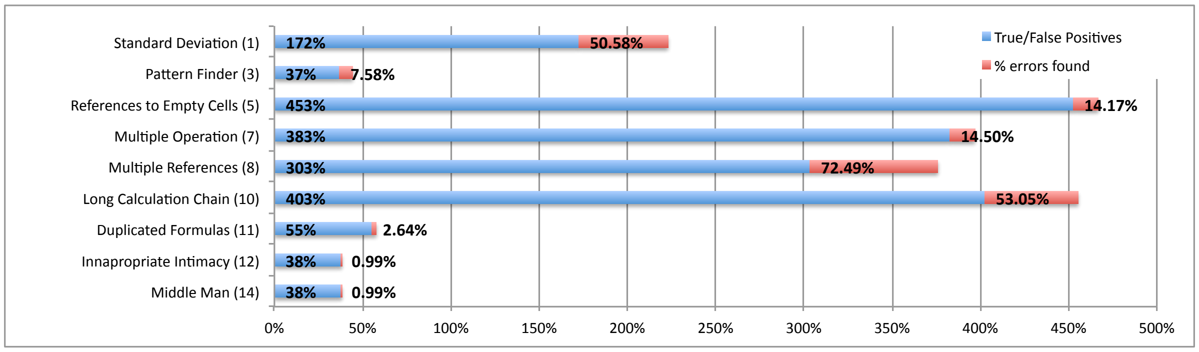
A second threat is the fact that we use only one corpus of spreadsheets. Although this is the case, it is also true that it is the largest corpus of spreadsheets with their errors documented, and it was created with the goal of providing a research corpus for spreadsheet error detection. This is a necessary 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.

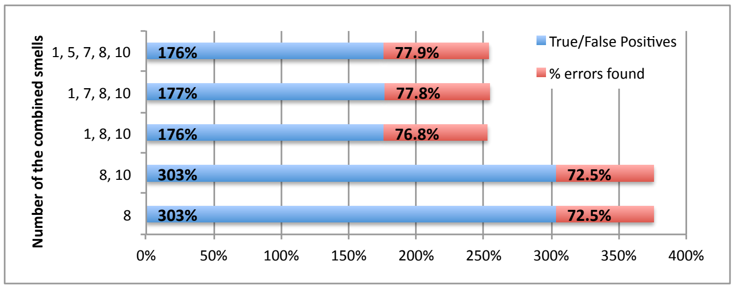
VIII. Related Work

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 [23], [24]. 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 [25].

GoalDebug [26] is a spreadsheet debugger targeted at end users. Whenever the computed output of a cell is incorrect, the



(a) True positive over false positives and percentage of errors found on single smells.



(b) Combination of 1 up to 5 of the smells with better results.

Fig. 4: Analysis of the behavior of smells individually and combined.

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.

There are several spreadsheet analysis tools that try to find inconsistencies in spreadsheet formulas [27], [28], [29], [30], [31], [32], 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 [33], which can perform unit analysis automatically by exploiting header inference techniques [27].

Other approaches that aim at minimizing the occurrence of errors in spreadsheets include code inspection [34], refactoring [10] and the adoption of better spreadsheet design practices [35], [36], but none of these approaches focuses on spreadsheets' debugging.

IX. 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 in 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 good smells, the first set, 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 [37].

References

[1] R. R. Panko and N. Ordway, "Sarbanes-oxley: What about all the spreadsheets?" *CoRR*, vol. abs/0804.0797, 2008. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr0804.html#abs-0804-0797

[2] F. Hermans, M. Pinzger, and A. van Deursen, "Supporting professional spreadsheet users by generating leveled dataflow diagrams," in *Proc. of the 33rd Int. Conf. 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

[3] R. Panko, "Spreadsheet errors: What we know. what we think we can do." *Proceedings of the 2000 European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.

[4] ——, "Facing the problem of spreadsheet errors," *Decision Line, 37(5)*, 2006.

[5] C. M. Reinhart and K. S. Rogoff, "Growth in a time of debt," *American Economic Review*, vol. 100, no. 2, pp. 573–78, September 2010. [Online]. Available: http://www.aeaweb.org/articles.php?doi=10.1257/aer.100.2.573

[6] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner, "On the empirical evaluation of fault localization techniques for spreadsheets," in *Proc. of the 16th Int. Conf. on Fundamental Approaches to Software Engineering*, 2013, pp. 68–82.

[7] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *ICSE*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE, 2012, pp. 441–451.

[8] ——, "Detecting code smells in spreadsheet formulas," in *ICSM*. IEEE, 2012, pp. 409–418.

[9] J. Cunha, J. P. Fernandes, J. Mendes, and J. S. Hugo Pacheco, "Towards a Catalog of Spreadsheet Smells," in *The 12th International Conference on Computational Science and Its Applications*, ser. ICCSA'12, vol. 7336. LNCS, 2012, pp. 202–216.

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

[11] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. of Systems and Sw.*, vol. 82, no. 11, pp. 1780–1792, 2009.

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

[13] S. Aurigemma and R. R. Panko, "The detection of human spreadsheet errors by humans versus inspection (auditing) software," in *Proc. of EuSpRIG Conf.*, 2010.

[14] J. Cunha, J. P. Fernandes, J. Mendes, P. Martins, and J. Saraiva, "Smellsheet detective: A tool for detecting bad smells in spreadsheets," in *Proc. of the 2012 VL/HCC*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 243–244.

[15] F. Chiang and R. J. Miller, "Discovering data quality rules," *The Proceedings of the VLDB Endowment.*, vol. 1, pp. 1166–1177, August 2008.

[16] E. F. Codd, "A relational model of data for large shared data banks." *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[17] R. Abreu, P. Zoeteweij, and A. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques – Mutation (Mutation'07)*, 2007, pp. 89–98.

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

[19] R. Abreu, P. Zoeteweij, and A. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, 2006, pp. 39–46.

[20] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[21] A. da Silva Meyer, A. Garcia, A. de Souza, and C. de Souza, "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l.)," *Genetics and Molecular Biology*, vol. 27, pp. 83–91, 2004.

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

[23] A. Bregar, "Complexity metrics for spreadsheet models," *Proceedings of the 2004 EuSpRIG Conf.*, vol. CoRR abs/0802.3895, pp. 85–93, 2004.

[24] K. Hodnigg and R. T. Mittermeir, "Metrics-based spreadsheet visualization: Support for focused maintenance," *CoRR*, vol. abs/0809.3009, 2008.

[25] 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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 292–301. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486827

[26] R. Abraham and M. Erwig, "Goaldebug: A spreadsheet debugger for end users," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 251–260.

[27] ——, "Header and unit inference for spreadsheets through spatial analyses," *Proc. of 2004 VL/HCC*, pp. 165–172, Sept. 2004.

[28] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi, "A type system for statically detecting spreadsheet errors," in *ASE*. IEEE CS, 2003, pp. 174–183.

[29] M. Erwig and M. Burnett, "Adding apples and oranges," *4th Int. Symp. on Practical Aspects of Declarative Languages*, pp. 173–191, 2002.

[30] R. Abreu, A. Riboira, and F. Wotawa, "Debugging Spreadsheets: A CSP-based Approach," in *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, 2012, pp. 159–164.

[31] R. Abreu, B. Hofer, A. Perez, and F. Wotawa, "Using constraints to diagnose faulty spreadsheets," *Software Quality Journal*, pp. 1–26, 2014.

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

[33] R. Abraham and M. Erwig, "UCheck: A spreadsheet type checker for end users." *J. Vis. Lang. Comput.*, vol. 18, no. 1, pp. 71–95, 2007.

[34] R. R. Panko, "Applying code inspection to spreadsheet testing," *Journal of Management Information Systems*, vol. 16, no. 2, pp. 159–176, Fall 1999.

[35] 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. IEEE CS, 2010, pp. 93–100.

[36] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, "MDSheet: A Framework for Model-driven Spreadsheet Engineering," in *Proceedings of the 34rd International Conference on Software Engineering*, ser. ICSE '12. ACM, 2012, pp. 1412–1415.

[37] R. Abraham and M. Erwig, "Mutation operators for spreadsheets," *IEEE Trans. Software Eng*, vol. 35, no. 1, pp. 94–108, 2009. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.73