FaultySheet Detective: When Smells Meet Fault Localization

Rui Abreu*, Jácome Cunha†‡, João Paulo Fernandes†§, Pedro Martins†, Alexandre Perez*, João Saraiva†
* Departamento de Engenharia Informática, FEUP, Portugal
rui@computer.org,alexandre.perez@fe.up.pt
† HASLab/INESC TEC & Universidade do Minho, Portugal
{jpaulo,prmartins,jas}@di.uminho.pt
‡ Universidade Nova de Lisboa, Portugal
jacome@fct.unl.pt
§ RELEASE, Universidade da Beira Interior, Portugal

***Abstract*—This paper presents a tool, dubbed *FaultySheet Detective*, for aiding in spreadsheet fault localization, which combines the detection of bad smells with a generic spectrum-based fault localization algorithm.**

***Keywords*—Spreadsheets; Smells; Errors; Fault Localization; FaultySheet Detective.**

I. Introduction

Spreadsheet systems have achieved an astonishing success in terms of both the number of their users as well as the variety of domains in which they are used. This importance, however, has not been achieved together with effective mechanisms for error prevention, as shown by several studies [1], [2], and a long list of horror stories with real impacts[1].

We therefore believe that spreadsheet fault localization[2] techniques are much needed. In fact, the natural trend of incorporating well-established programming language features under spreadsheets has been witnessed again by the integration of spectrum-based fault localization methods under a spreadsheet system [3] and the identification of spreadsheet *bad smells* [4], [5], [6], [7], while these techniques are well established for traditional programming languages: [8] and [9], respectively.

In this paper we present the *FaultySheet Detective* tool that combines bad smells and fault localization techniques to create a debugging framework for spreadsheets. This tool implements the full state-of-the-art catalog of spreadsheet smells: we cover all spreadsheet smells published in the literature. Our method initially builds on the suspicion-raising nature of smells to identify spreadsheet cells that may be of harm. Secondly, a particular subset of all cells identified by smells is provided as input to a fault localization algorithm. This step is realized as an attempt to identify spreadsheet cells that may have contributed to a cell being considered a suspect. The combination of bad smells and fault localization techniques used by FaultySheet Detective is described in detail in [10].

Empirical experiments using the toolset, thoroughly reported and discussed in [10], using a well-known faulty spreadsheet catalog, have shown that our approach is able to identify more than 70% of errors in spreadsheets in a setting where two out of three identified faulty cells are errors.

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

[2]In this paper, we also use the term debugging.

II. Smells Meet Fault Localization

In this section we briefly introduce the concepts of spreadsheet smells, fault localization for spreadsheets, and how these two techniques can be combined to find faults in spreadsheets.

A. Spreadsheet Smells

The concept of *code smell* was introduced by Martin Fowler as a first symptom that may correspond to a deeper problem in a system [9]. This definition holds two subtleties: i) a smell must be easy to spot, e.g., in an object-oriented language, a method with over a dozen arguments may immediately be spotted; ii) a smell does not always imply an error: a method with twenty arguments may be free of problems. Fowler also proposed an initial catalog of potential problems in the form of smells. This catalog was originally defined for source code, but Fowler's work inspired several authors to propose different catalogs of smells for spreadsheets [6], [11], [5], [4]. We have taken the union of all the proposed catalogs, obtaining the comprehensive list that we review next.

1. *Standard Deviation*: Detects cells not following the normal distribution of a group of cells holding numerical values.

2. *Empty Cell*: Cells that are left empty but that occur in a context that suggests they should have content are detected.

3. *Pattern Finder*: Finds broken patterns, e.g. a row containing only numerical values except for one cell holding a formula.

4. *String Distance*: Signals string cells that differ minimally with respect to other surrounding cells.

5. *Reference to Empty Cells*: Formulas pointing to empty cells are detected.

6. *Quasi-Functional Dependencies*: Occurs when equal values in a column correspond to the same values in another column, except for a few cases.

7. *Multiple Operations*: Flags formulas with many operations.

8. *Multiple References*: When a formula references many different cells this smell is raised.

9. *Conditional Complexity*: Detects formulas with many conditional operations.

10. *Long Calculation Chain*: Detects formulas with long calculation chains.

11. *Duplicated Formulas*: Indicates that similar snippets of code are used throughout different cells.

12. *Inappropriate Intimacy*: Recognizes a worksheet that is too much related to a second one.

13. *Feature Envy*: Appears when a formula is more interested in the cells of a worksheet other than the one that contains it.

14. *Middle Man*: Recognized if a 'middle man' formula contains only a reference to another cell and calculations.

15. *Shotgun Surgery*: Occurs when a formula is referred by many different formulas in different worksheets.

FaultySheet Detective starts by automatically processing spreadsheets and determining their *smelly* cells. A subset of these cells is then selected, and automatically feeds a fault localization process. The purpose of the next subsection is to revise spectrum-based fault localization for spreadsheets.

B. Spectrum-based Fault Localization for Spreadsheets

Spectrum-based Fault Localization (SFL) is a software debugging technique that calculates the likelihood of each system component (*e.g.*, each statement) being faulty [12]. This technique exploits coverage information from passed an failed system runs. The information gathered from these runs is their hit spectra matrix, which consists of a set of flags for every component that represent whether they were touched or not by each execution. The fault localization consists in identifying what components resemble the passed and failed runs the most. For instance, if a component was only executed when the system failed, it has a higher likelihood of containing a fault when compared to a component that was only executed on passed executions. Similarity coefficients, like the Ochiai coefficient [13], are calculated for every component and ranked to yield the diagnostic report.

In order to use SFL for spreadsheet fault localization, some modifications need to be performed [3]. For instance, the concept of code coverage does not translate directly to the spreadsheet paradigm. As an alternative to code coverage, the *cone* of a cell (*i.e.*, a set of a cell and its data dependencies) can be computed. The set of *cones* for every output cell (a cell that is not referenced by any other cell) is essentially the hit-spectra matrix needed to perform fault localization.

C. Smells Meet Fault Localization

FaultySheet Detective combines both techniques just described. It first detects all the smells contained in a spreadsheet. These smelly cells are then passed as arguments to the fault localization algorithm. In fact it is possible to select which smells are detected, thus restricting the cells sent to the fault localization algorithm. Moreover, it is also possible to define the threshold of the number of smells shown, that is, the user can decide to see only the cells containing the higher number of smells, the cells with the higher number of smells and the ones with the higher less one, etc.

III. The SmellSheet Detective Framework

In this section we describe *FaultySheet Detective*. In the past we have introduced another related, but it could only compute seven smells and did not use the algorithms for fault localization [11], thus being much more limited than *FaultySheet Detective*.

A. Architecture

FaultySheet Detective supports both desktop spreadsheets written in Excel and spreadsheets developed in the Google Drive platform. The support for online spreadsheets was added motivated by the fact that the migration from desktop to online-based applications is becoming very common, and even the popular Microsoft Office suite has its online version.

The implementation of this tool combines the Java programming language, the Apache POI library, and the Google Data API's to work with spreadsheets within the Google Drive environment. Figure 1 illustrates the architecture of *FaultySheet Detective* and in the following paragraphs we will explain in detail each part of this design.

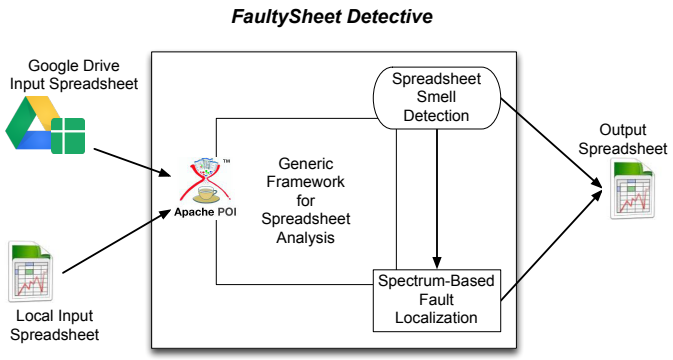

Fig. 1. *FaultySheet Detective* architecture.

a) Apache POI: Apache POI is a Java library that manipulates spreadsheet artifacts[3]. It allows to read a spreadsheet file and store it in an object with several useful methods. It also allows to export one spreadsheet object to a spreadsheet file. We use this library as support for our Generic Framework for Spreadsheet Analysis. We also use it to create the new spreadsheet annotated with the faults detected.

b) Google GData APIs: The Google GData API's[4] provide external access to data and functionality through the Google Data Protocol to various Google services. We use it to access Google Drive accounts.

c) Generic Framework for Spreadsheet Analysis: Although Apache POI is a very interesting tool to manipulate spreadsheets, while implementing *FaultySheet Detective*, we felt the need to abstract certain methods to allow Spreadsheet operations to be easier and more analysis-oriented. Therefore, we developed a generic framework to analyze spreadsheets where functionalities such as traversing all the cells or retrieve

[3]In fact, this library can manipulate other kinds of documents. More details about Apache POI can be found in http://poi.apache.org.

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

cells information is easier than using Apache POI. Since we believe this framework would be useful to others, we also make it available as as Java standalone library, which can be accessed in http://ssaapp.di.uminho.pt.

d) Detection of Smells: At this point we are able to define specific methods to detect the necessary smells. For each smell in the literature, we implemented a method so it can be detected. Given the extensibility feature of our software, adding it with new smells is very simple. For this, it is only necessary to write a Java method implementing its detection. The tool will then adapt itself to such a new smell. This is very important when developing a catalog of smells, since new smells can easily be considered and added.

e) Spectrum-based Fault Localization: The list of detected smells is fed into the fault localization framework detailed in section II-B. Prior to executing the SFL technique, the *cones* for all cells are calculated, as well as output cells. Output cells are simply given by subtracting the set of cells inside a *cone* to the list of all cells. After performing the fault localization, every cell is given a similarity score, in our case using the Ochiai similarity coefficient, that quantifies the likelihood of that cell being at fault. Higher coefficients are more likely to contain a fault than lower coefficients.

f) Combining Smells with SFL: We can now explain how all these parts are combined to produce a spreadsheet annotated with detected faults. We use a matrix were each entry corresponds to a cell from the spreadsheet being analyzed. Each entry of this matrix is an integer and is initialized with the value 0. Then, for each smelly cell detected, the corresponding weight is added to the correct matrix entry. After all smells have been detected, the cells that fit the threshold defined are then sent to the SFL algorithm. This threshold can be set to 0, meaning only the cells with the highest number of smells detected are arguments to SFL, 1, meaning that the cell with the maximum number of smells, plus the cells with the maximum minus one unit, are passed, and so on. This value is by default set to 2 since this have the best results in the studies we conducted [10]. These cells are then passed to the SFL algorithm and the cells it reports are incremented in the matrix by the corresponding weight. From this matrix, the tool colors the corresponding cells and adds the necessary notes with the detected problems (we discuss in the next paragraph this in more detail). A new spreadsheet file is then outputted to the same folder where the original file was read from. In case a spreadsheet from Google Drive has been chosen, the path where the tool is being run is the default folder.

g) Annotating the Resulting Spreadsheet: FaultySheet Detective computes a new spreadsheet that is a copy of the original one, but with some cells' background colored. Each cell where faults are found gets its background colored depending corresponding matrix entry. The color goes from a green very clear to a very dark red: the greater the number of faults, the greater red the color will be. This technique has also been used in many other works were it is necessary to give some feedback to the users [5], [14], [15], [16], [17]. Figure 2 illustrates the correspondence between the number of faults detected and the color of the background a cell will get.

Users should tackle first the cells that have a more alarming color, that is, the cells that are more red in the color scale

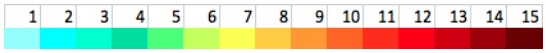

Fig. 2. Correspondence between the number of faults and the color a cell gets its background.

shown. The only situation where colors can be problematic is when the spreadsheet itself already contains colors. Although this is a good practice to use in spreadsheets, unfortunately it does not happen very often.

Besides the colors, a note is attached to each cell were faults are detected . Our tool writes the smells that were detected and the result of SFL in that cell. The user can then try to improve the cell design by tackling each of the issues raised. In fact, several refactorings have been proposed to solve such issues [5], [4], [7]. Since these notes can have as much information as necessary to improve the feedback sent to user, we plan to integrate suggestions to the user using the refactorings proposed. Moreover, we believe that with more research we could rank the issues raised so the user knows which ones should work on first. This technique of creating some kind of helper text to give feedback to users as also been proposed by other authors [5], [4].

B. Usage

When using the Google Drive's variant of our tool, a valid Google Drive account is required, as we can see in Figure 3. On the other hand, when using analyzing local spreadsheets, the user can browse his/her local machine. For now, *FaultySheet Detective* allows the analysis of a single spreadsheet at a time, but it can easily be adapted to run on a folders, for example.

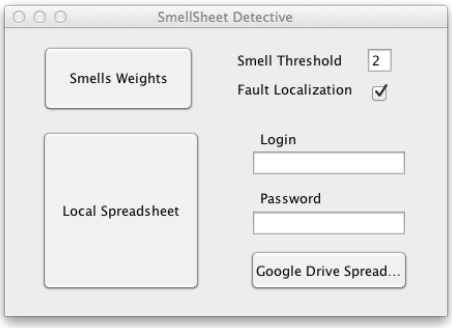

Fig. 3. *FaultySheet Detective* user interface.

Although the smell weights and thresholds are already defined, users can adapt them to their needs, as shown in Figure 4. Moreover, users can also disable the use of SFL, in case he only wants to search for bad smells in the Spreadsheet without searching for further dependencies using SFL.

Figure 5 shows an example of a spreadsheet that has been analyzed by *FaultySheet Detective*. Note that when the user passes the mouse over a cell that is colored by our tool, the spreadsheet system will show the content of the corresponding note, that is, it will show the smells detected. For now the only result the tool is producing is a new spreadsheet copied from the original one and colored according to the potential errors found. We can easily compute different outputs, such as a

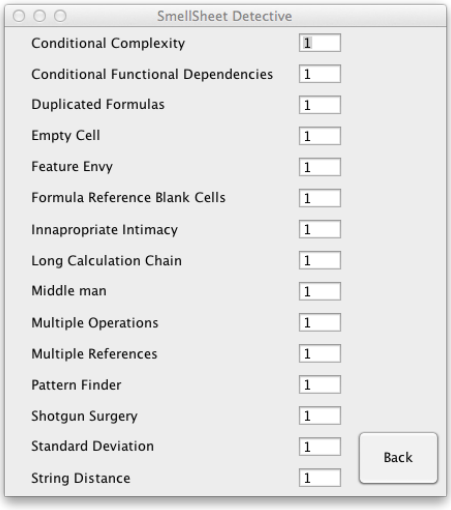

Fig. 4. *FaultySheet Detective* user interface to manipulate weights.

report describing the possible errors found in the spreadsheets. In fact, we plan to implement this feature.

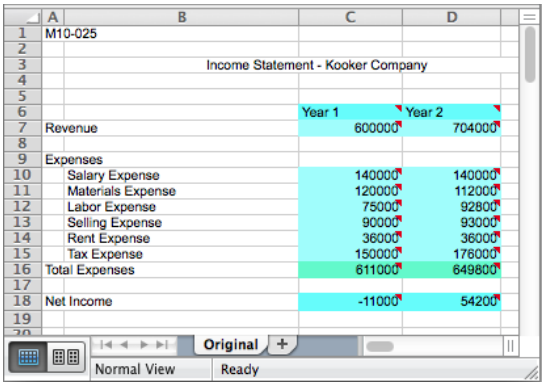

Fig. 5. Example of a spreadsheet being analyzed by *FaultySheet Detective*.

Availability: *FaultySheet Detective* and a video demonstrating its use are available at http://ssaapp.di.uminho.pt (Software page).

IV. Related Work

This section presents other tools that focus on spreadsheet debugging. GoalDebug [18] 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. The tool generates list of change suggestions for cell formulas that, when applied, would result in the user-specified output. Users are expected to detect errors manually, and provide the system with the correct output value. In our tool, the error detection is automated by testing spreadsheets against a smell catalog.

Breviz [4] is a tool that locates inter-worksheet smells in spreadsheets, and presents them via data flow diagrams, to improve understanding. UCheck [14] tries to find inconsistencies in spreadsheet formulas. It does not require the user to annotate cell with additional information, as the analysis is performed by exploiting header inference techniques.

MDSheet is a framework aimed at minimizing the occurrence of errors in spreadsheet through the adoption of better spreadsheet design practices [19]. This approach, however, does not focus on the debugging of spreadsheets.

V. Conclusion

This paper describes *FaultySheet Detective*, a tool for automatic fault localization in spreadsheets. It uses a catalog of fifteen spreadsheet smells to provide an indication of possible faults, which are then fed into a well established SFL algorithm to provide a diagnosis. Experiments show that two out of three identified faulty cells are documented errors.

Future work includes devising intuitive visualizations to convey the diagnostic data. Furthermore, we plan to provide fix suggestions for detected errors and to extend the spreadsheet smell catalog with more subjects.

References

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

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

[3] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner, "On the empirical evaluation of fault localization techniques for spreadsheets," in *FASE'13*, 2013, pp. 68–82.

[4] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *ICSE'12*, 2012, pp. 441–451.

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

[6] J. Cunha, J. P. Fernandes, J. Mendes, and J. S. Hugo Pacheco, "Towards a Catalog of Spreadsheet Smells," in *ICCSA'12*, 2012, pp. 202–216.

[7] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *ICSM'12*, 2012, pp. 399–409.

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

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

[10] R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez, and J. Saraiva, "Smelling faults in spreadsheets," in *ICSME '14*, 2014.

[11] J. Cunha, J. P. Fernandes, J. Mendes, P. Martins, and J. Saraiva, "Smellsheet detective: A tool for detecting bad smells in spreadsheets," in *VLHCC'12*, 2012, pp. 243–244.

[12] R. Abreu, P. Zoeteweij, and A. van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART – Mutation'07)*, 2007, pp. 89–98.

[13] ——, "An evaluation of similarity coefficients for software fault localization," in *PRDC'06*, 2006, pp. 39–46.

[14] 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.

[15] C. Chambers and M. Erwig, "Automatic detection of dimension errors in spreadsheets," *J. Vis. Lang. Comput.*, vol. 20, no. 4, pp. 269–283, 2009.

[16] M. Erwig, "Software Engineering for Spreadsheets," *IEEE Software*, vol. 29, no. 5, pp. 25–30, 2009.

[17] R. Abraham and M. Erwig, "How to communicate unit error messages in spreadsheets," in *WEUSE I*, 2005, pp. 1–5.

[18] ——, "Goaldebug: A spreadsheet debugger for end users," in *ICSE'07*, 2007, pp. 251–260.

[19] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, "MDSheet: A Framework for Model-driven Spreadsheet Engineering," in *ICSE'12*, 2012, pp. 1412–1415.