A Decomposition-Based Approach to Spreadsheet Testing and Debugging

Thomas Schmitz*, Dietmar Jannach*, Birgit Hofer†, Patrick Koch†,
Konstantin Schekotihin‡, and Franz Wotawa†
*TU Dortmund, Germany, Email: {firstname.lastname}@tu-dortmund.de
†Graz University of Technology, Austria, Email: {bhofer, koch, wotawa}@ist.tugraz.at
‡Alpen-Adria Universität Klagenfurt, Austria, Email: konstantin.schekotihin@aau.at

***Abstract*—Spreadsheets serve as a basis for decision-making processes in many companies and bugs in spreadsheets can therefore represent a considerable risk to businesses. Systematic tests can help to locate such bugs, but providing test cases can be cumbersome and complex for large real-world spreadsheets.**

To make the specification of test cases easier, we propose to split spreadsheets into smaller logically connected parts (called fragments) which can be individually tested for correctness. We present an algorithmic approach to compute such fragments, which we validated with a laboratory study in the form of a spreadsheet debugging exercise involving 57 subjects. The results show that the fragmentation approach can help to significantly reduce the required efforts to test a spreadsheet.[1]

I. Introduction

Spreadsheets are widely used in companies for everyday decision making processes since they can be built and maintained by almost everybody familiar with the business logic. However, spreadsheets developed by end users also have a high risk of containing faults. Many examples show that these faults can have severe impacts for companies [2], [3].

Many approaches for finding and localizing faults in spreadsheets were proposed in the literature [4]. One possible way to find faults is to *test* the spreadsheet in a systematic way [5]–[7] by providing *test cases* – sets of input and expected output values. A recent study suggests that professional spreadsheet users indeed test their spreadsheets in some form or another [8]. However, for large spreadsheets, the specification of test cases can be difficult and error-prone, because the user has to manually find inputs and calculate outputs that result in the identification of a fault.

To solve this problem one can decompose a given spreadsheet into smaller logically connected parts and let the user specify test cases for these smaller "fragments". It allows the user to focus on the (much shorter and fewer) calculation chains within the fragment, which in turn should lead to lower cognitive load for the end user when specifying the test cases. The fragmentation of a spreadsheet can be done manually or through an automated process [1].

In this paper, we propose a method to automatically compute a "good" fragmentation of a given spreadsheet. Our approach uses heuristics and a genetic algorithm to find a balance between the number of resulting fragments and their average estimated complexity for the end user. The implemented fragmentation technique is embedded within a comprehensive test and debugging environment that is designed as an add-in for the Microsoft Excel spreadsheet environment.

We validated our approach with a between-subjects user study in the form of a spreadsheet testing exercise, where the task of the participants was to fully test a given spreadsheet in different conditions. The participants of the study either (a) used the automated fragmentation approach proposed in this paper, (b) created the fragments for testing manually, or (c) tested the spreadsheet without using any fragments at all.

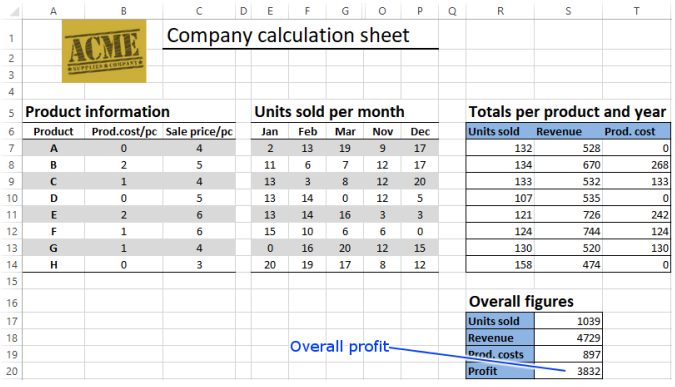
[1]A previous version of the fragmentation algorithm proposed in this work was presented at the SEMS 2016 workshop [1].

978-1-5386-0443-4/17/$31.00 ©2017 IEEE

II. Motivating Example

The example spreadsheet in Figure 1 (adapted from [9]) contains a typical profit calculation of a fictitious company over the course of a year. It has three main parts:

1) The required inputs to the calculation (column A to column P) are (a) the production costs and sales prices for the different products and (b) the monthly sales numbers.
2) These input values are aggregated for each product through different calculations on the right-hand side of the spreadsheet in columns R, S, and T (row 7 to 14).
3) The totals per product are then combined to a profit summary for the entire year in column S (row 17 to 20).

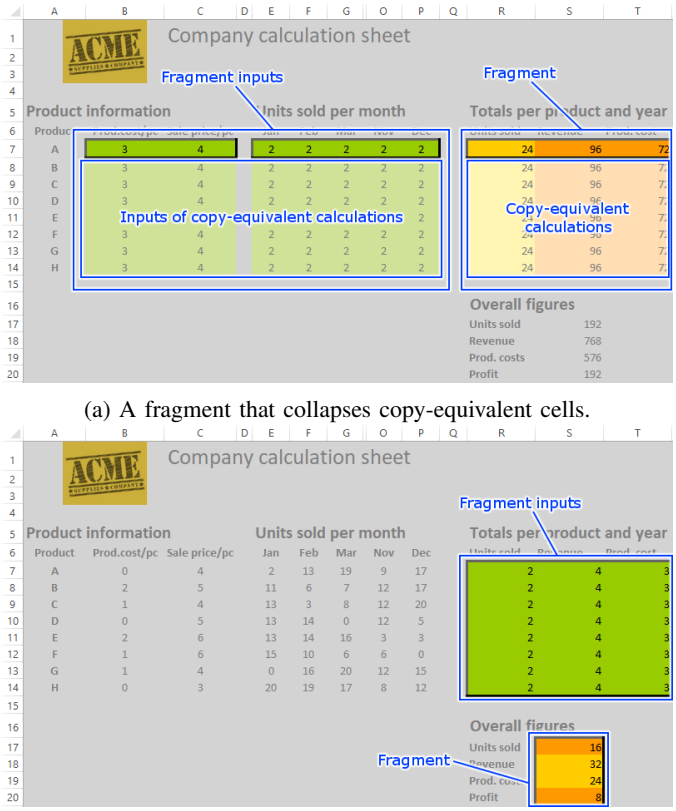


Product	Prod.cost/pc	Sale price/pc	Jan	Feb	Mar	Nov	Dec	Units sold	Revenue	Prod. cost
A	0	4	2	13	19	9	17	132	528	0
B	2	5	11	6	7	12	17	134	670	268
C	1	4	13	3	8	12	20	133	532	133
D	0	5	13	14	0	12	5	107	535	0
E	2	6	13	14	16	3	3	121	726	242
F	1	6	15	10	6	6	0	124	744	124
G	1	4	0	16	20	12	15	130	520	130
H	0	3	20	19	17	8	12	158	474	0

Overall figures	
Units sold	1039
Revenue	4729
Prod. costs	897
Profit	3832

Fig. 1: Example spreadsheet of a profit calculation. [9]

The spreadsheet is of still manageable complexity and one can find much more complex ones in the real world. When



(a) A fragment that collapses copy-equivalent cells.

(b) A fragment containing neighboring cells.

Fig. 2: Fragmentation of the example spreadsheet.

adopting a testing-based approach the user has to check, e.g., the correctness of the value in cell $S20$ (the overall profit). However, checking if the observed value is correct can be tedious, even for small spreadsheets, due to the potentially long calculation chains from the input to the output cells.

Figure 2 shows an outcome of our fragmentation method, which decomposes the spreadsheet into two fragments. The first one, shown in Figure 2a, contains the cells from R7 to T14 computing the totals per product and year. This fragment uses the cells from B7 to P14 as inputs, i.e., the inputs of the original spreadsheet. Our Excel add-in visually highlights the cells of the fragment: input cells are green, intermediate calculations are yellow, output cells are orange, and cells that do not belong to the fragment are greyed out. A lesser intensity of the color shows that the calculations in these cells are "copy-equivalent", i.e., they do the same calculations but use different inputs.

In our approach, the user only has to provide test cases for the highlighted areas of the individual fragments. Given the limited number of inputs and outputs and the shorter calculation chains, this is much easier than providing test cases for the entire spreadsheet. Furthermore, in the case of copy-equivalent calculations, test cases only have to be provided for one of the copies.

The second fragment is shown in Figure 2b and comprises the calculations for the overall profitability numbers. Our approach ends up with this fragment due to (a) dependencies between the cells and (b) their position. The system highlighted the final calculations in the cells $S17$ to $S20$ as well as the intermediate calculations in the range $R7$ to $T14$, which contain formulas. In the context of the fragment, they are however considered as input cells and are marked with green color. Therefore, when the user specifies a test case for the fragment, he or she provides arbitrary input values for these cells and focuses on checking the correctness of the outputs in the cells $S17$ to $S20$. As a result, the calculation chains become much shorter. In addition, when the user provides small values, the manual checks for the correctness of the outputs become much easier.

III. An Automated Fragmentation Approach

As discussed in the previous section, splitting a complex spreadsheet into fragments helps us to focus on a subset of the calculations when specifying a test case. Therefore, we aim at cells that contain calculations (formulas).

Definition 1 (Fragment and Fragmentation)**.** *Let S be a spreadsheet (a set of cells) and $formulaCells : S \mapsto S_f$ be a function that maps a set of cells S to its subset S_f consisting of all cells with formulas in S. A* fragment *f of S is a set of cells with formulas, i.e., $f \subseteq formulaCells(S)$. A* fragmentation *$F$ is a set of fragments of S. F is a* covering *fragmentation if $formulaCells(S) = \bigcup_{f_i \in F} f_i$.*

In our approach we are usually interested in a *covering* fragmentation in order to cover all possible causes when a spreadsheet's calculation outcomes are not as expected.

Example 1. *The fragmentation F of the example spreadsheet shown in Figure 2, $F = \{\{R7, S7, T7, R8, \ldots, T14\}, \{S17, S18, S19, S20\}\}$, is covering, as every cell containing a formula is part of at least one fragment.*

Our automated fragmentation approach has two phases, which we explain in the following subsections.

A. Detecting Copied Formula Cells

To simplify the test process our approach automatically detects areas with semantically equivalent calculations as shown in Figure 2a. Once such areas are detected, the user is only required to check the correctness of a smaller part (i.e., one copy of the calculations) of the fragment. If this part is considered error-free by the user, the other areas that contain identical calculations are assumed to be error-free, too. Similarly, if there is an error in one of the formulas of this part, it has been copied to the other areas as well.

The automatic detection of semantically related areas in our approach relies on the following heuristics. First, the calculations in the respective areas must be *copy-equivalent*. Two formula cells are copy-equivalent, if their formulas are identical when using the relative R1C1 formula notation. Second, we restrict the detection of copy-equivalent cells only to *column-row-related* fragments.

Definition 2 (Column-Row-Relatedness)**.** *A fragment f is* column-row-related*, if there exists a connected graph $G = (f, E)$, where every cell $c \in f$ corresponds to a node and the set of edges $E \subseteq f \times f$ connects the nodes such that $\forall (c, c') \in E : x(c) = x(c') \vee y(c) = y(c')$, where $x(c)$ denotes the column of c and $y(c)$ its row.*

Given this definition, we collapse all cells that are assumedly copies of each other in what we call a *base fragment* and define a representative cell for it. In the example in Figure 2a, the cells $R7$ to $R14$ contain identical calculations and represent such a base fragment. The cell $R7$, which is the top-most cell of the base fragment and has the smallest column index is chosen as a representative.

Definition 3 (Base Fragment)**.** *A column-row-related fragment f_b is called a* base fragment*, if the condition $\forall c, c' \in f_b$: copy-equivalent(c, c') holds for f_b, but does not hold for any column-row-related fragment $f'_b \supset f_b$. The left-most cell c in the top-most row of a base fragment f_b is called the* representative cell *of f_b and is returned by the function representativeCell$(f_b) : f_b \mapsto c$, where c satisfies the condition $\forall c' \in (f_b \setminus \{c\}) : y(c) < y(c') \vee (y(c) = y(c') \wedge x(c) < x(c'))$.*

A base fragment therefore either comprises a single formula cell or a set of formula cells with calculations that are assumed to be semantically equivalent.

Example 2. *The example spreadsheet shown in Figure 1 has seven base fragments: {R7, R8, ..., R14}, {S7, S8, ..., S14}, {T7, T8, ..., T14}, {S17}, {S18}, {S19}, and {S20}.*

B. A Bottom-Up Fragment Generation Procedure

The goal of our approach is to find a fragmentation that is *optimal* for the end user in terms of its complexity. We estimate the complexity of a fragmentation with different heuristics, e.g., the average size of the fragments, the complexity of the formulas, and the overall number of fragments. These heuristics are based on the ideas of code smells [10], [11] and spreadsheet complexity measures [12].

When computing the complexity of a given fragmentation, we only consider the representatives of the base fragments.

Definition 4 (Representatives)**.** *Let $f = \{f_{b1}, \ldots, f_{bn}\}$ be a fragment consisting of n base fragments. The function* Rps(f) *returns the set of representative cells of the base fragments:*

$$Rps(f) \coloneqq \bigcup_{f_{bi} \in f} representativeCell(f_{bi}).$$

To calculate the complexity of an individual fragment we use four heuristic complexity metrics.

$$H_{in}(f) \coloneqq |input(Rps(f))| \tag{1}$$

$$H_{out}(f) \coloneqq |output(Rps(f))| \tag{2}$$

$$\begin{aligned} H_{area}(f) \coloneqq\ & (max_x(Rps(f)) - min_x(Rps(f)) + 1) \\ & * (max_y(Rps(f)) - min_y(Rps(f)) + 1) \end{aligned} \tag{3}$$

$$H_{formulas}(f) \coloneqq \sum_{c \in Rps(f)} formulaComplexity(c) \tag{4}$$

The function *input* returns the number of input cells for a fragment, *output* the number of output cells, max_x returns the index of the last column for a set of cells and min_x the index of the first column. The function *formulaComplexity* assesses the complexity of a cell's formula by the number of conditionals and cell references in it.

The complexity heuristics lead to the following desired effects. Minimizing the number of inputs and outputs means that merging fragments is favorable when they have the same input and output cells (Heuristics H_{in} and H_{out}). Heuristic H_{area} favors fragments that contain "physically" close cells over fragments that cover distant cells. Heuristic $H_{formulas}$ finally penalizes fragments that contain too many complex formulas. Other heuristic complexity measures, such as the length of the calculation chains, can easily be added to our framework. However, when using, e.g., calculation chain lengths, one has to assess if a fragment consisting of short chains with nested formulas is not more complicated to test for users than one with long chains but simple formulas.

Generally, the different aspects that are encoded in the heuristics are not necessarily equally important and the complexity heuristics do not use the same scales. Therefore, we propose to weight the different complexity values before we sum them up for each fragment.

$$fC(f) \coloneqq \sum_{i=0}^{|H|} H_i(f) * w_i \tag{5}$$

where H is a list of all implemented heuristics and w a vector containing the weights of the heuristics. The weights can be set manually or with the help of some optimization technique. In our tests, we set the weights to $w = (0.2, 1, 1, 1)$.

The complexity estimates for the fragments and the number of fragments are then used to determine the fitness of a fragmentation. To ensure that a fragmentation does not contain a few very complex and some simple fragments, we furthermore factor the standard deviation of all fragment complexity estimates into the fitness function. The final fitness function is as follows.

$$fitness(F) \coloneqq -\left(\sum_{f \in F} fC(f)\right) - |F| * w_F - \sigma(F) * w_\sigma \tag{6}$$

where w_F is a weight factor to adjust the penalty for fragmentations with a large number of fragments, $\sigma(F)$ is the standard deviation, and w_σ is another weight factor to adjust the penalty for a large standard deviation. The standard deviation is defined as

$$\sigma(F) \coloneqq \sqrt{\frac{\sum_{f \in F} \left(fC(f) - \frac{\sum_{f \in F} fC(f)}{|F|}\right)^2}{|F|}}.$$

We set $w_F = 0.1$ and $w_\sigma = 0.2$ for our evaluations.

The outcome of (6) represents the fitness value of a given fragmentation, which corresponds to the inverse of its estimated complexity. The goal of the optimization process is to find the fragmentation that has the lowest complexity.

Algorithm 1: Fragment Generation

Input: A spreadsheet S, population size p, number of generations g, survival rate s, heuristic weights w, w_F, and w_σ

Output: A covering fragmentation F

```
1 B ← createBaseFragments(S);
2 P ← createInitialPopulation(B, p);
3 for i ∈ {1, ..., g} do
4     P ← selectFittestIndividuals(P, s, w, w_F, w_σ);
5     P ← P ∪ getMutants(P, p − |P|);
6 F ← selectFittestIndividual(P, w, w_F, w_σ);
7 return F;
```

Since there is an exponential number of fragment candidates for a spreadsheet, the computation of an optimal fragmentation might be challenging. Therefore, our system uses an evolutionary algorithm which creates the fragments in a bottom-up fashion. We start by identifying the base fragments as described above which we then incrementally merge with other fragments over the course of multiple generations. In each generation a number of fragmentations is created up to the maximum population size by merging or dividing the fragments of the other fragmentations. Of the generated fragmentations only the fittest ones "survive" their generation and are kept in the population. This is done until we arrive at a "good" or close-to-optimal fragmentation. Note that the optimality of solutions typically cannot be guaranteed in evolutionary algorithms.

Algorithm 1 summarizes the fragment generation process, which results in a covering fragmentation. The algorithm takes as inputs the set S of formula cells of the spreadsheet that should be fragmented, the population size p, i.e., the number of individuals that can exist at any time, the number of generations g, the percentage s of mutants that should survive in each generation, and the heuristic weights w, w_F, and w_σ.

The procedure *createBaseFragments(S)* (Line 1) takes as input a set of cells and creates the base fragments according to Definition 3. The resulting fragmentation B is stored as the base fragmentation and is used to create the initial population.

In the evolution step, the fittest individuals are selected. The function *selectFittestIndividuals* takes as input the population P, the selection rate s, and the heuristic weights w, w_F, and w_σ and computes the fitness value for each individual $F \in P$ according to (6). The fittest $s * p$ individuals are kept in the population. In Line 5 mutants are created until the number of individuals is again equal to the population size p. The mutants are generated by randomly merging fragments or dividing them into their base fragments. In the next iterations, the evolution and selection steps are repeated.

At the end of the process, in Line 6, the fittest mutant, i.e., the fragmentation with the smallest sum of its complexity values, number of fragments, and standard deviation, is returned.

IV. Evaluation

To evaluate the benefits of using fragments to test a spreadsheet, we conducted a user study involving 57 subjects. In this study the participants had to test a spreadsheet in order to find faults that had been manually inserted. To test the spreadsheet one group of the participants created test cases for the entire spreadsheet (CG); another group manually created fragments and used these fragments to test the spreadsheet (MF); the last group tested the spreadsheet based on the automatically generated fragments as described in Section III (AF).

The results of the user study showed that using automatically generated fragments is beneficial for the users. The participants of group AF were able to test the spreadsheet $17\,\%$ faster than the other groups and managed to find $17\,\%$ more faults than group MF. Although group AF found the same amount of faults as group CG, using the automatically generated fragments reduced the effort that was required by the users to mark cells as faulty by $73\,\%$. In addition, both groups that used fragments to test the spreadsheet stated in a post-study questionnaire that they preferred the automatically generated fragmentation method over manually creating the fragments.

V. Related Work

Various approaches were proposed over the years to support the users when testing and debugging spreadsheets [4].

A visual testing methodology where test cases are entered and visualized within the spreadsheet was for example presented in [5]. We adopt a similar approach and users of our system can enter test cases within their known environment and mark output cells as being faulty or correct. The approach proposed in [7] goes one step further and automatically generates input values for the test cases in a way that all calculation paths, e.g., in the case of if-formulas, are executed. This functionality would also be helpful for our fragment-based approach and would reduce the effort for the user. An alternative way to support users during testing was proposed in [13], where the authors developed a method to point users to formulas that should be tested. While the goal of their work is to increase the coverage of the test cases, our work aims at covering *all* formulas with test cases and our fragment generation approach makes sure that indeed all cells with formulas are part of a fragment.

VI. Conclusion

We have proposed and empirically validated an algorithmic approach to decompose complex spreadsheets into smaller fragments, with the goal to reduce the complexity and effort for the user during testing and debugging. Our future works include (i) the incorporation of additional spreadsheet-specific heuristics in the fitness function of our evolutionary approach, (ii) the investigation of alternative forms of creating the fragments (e.g., using clustering methods), and (iii) the exploration of methods to automatically provide test cases for the fragments using existing test case generation methods from the literature on general software engineering.

ACKNOWLEDGMENT

The work was funded by the Austrian Science Fund (FWF, contract I2144) and the German Research Foundation (DFG, contract JA 2095/4-1).

REFERENCES

[1] T. Schmitz, B. Hofer, D. Jannach, and F. Wotawa, "Fragment-based diagnosis of spreadsheets," in *Proceedings of the 3rd International Workshop on Software Engineering Methods in Spreadsheets (SEMS 2016)*, Vienna, Austria, 2016, pp. 372–387.

[2] G. Tan, "Spreadsheet mistake costs Tibco shareholders $100 million," published: 2014-10-16; last visited: 2017-02-27. [Online]. Available: http://on.wsj.com/1vjYdWE

[3] F1F9, "The Dirty Dozen," last visited: 2017-02-27. [Online]. Available: http://blogs.mazars.com/the-model-auditor/files/2014/01/12-Modelling-Horror-Stories-and-Spreadsheet-Disasters-Mazars-UK.pdf

[4] D. Jannach, T. Schmitz, B. Hofer, and F. Wotawa, "Avoiding, finding and fixing spreadsheet errors - a survey of automated approaches for spreadsheet QA," *Journal of Systems and Software*, vol. 94, pp. 129–150, 2014.

[5] G. Rothermel, L. Li, C. Dupuis, and M. Burnett, "What You See Is What You Test: A Methodology for Testing Form-Based Visual programs," in *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, Kyoto, Japan, 1998, pp. 198–207.

[6] M. Fisher, G. Rothermel, T. Creelan, and M. Burnett, "Scaling a Dataflow Testing Methodology to the Multiparadigm World of Commercial Spreadsheets," in *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE 2006)*, Raleigh, NC, USA, 2006, pp. 13–22.

[7] R. Abraham and M. Erwig, "AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006)*, Brighton, United Kingdom, 2006, pp. 43–50.

[8] S. Roy, F. Hermans, and A. van Deursen, "Spreadsheet testing in practice," in *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2017)*, Feb 2017, pp. 338–348.

[9] D. Jannach and T. Schmitz, "Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach," *Automated Software Engineering*, vol. 23, no. 1, pp. 105–144, 2016.

[10] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, Riva del Garda, Trento, Italy, 2012, pp. 409–418.

[11] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva, "Towards a catalog of spreadsheet smells," in *Proceedings of the 12th International Conference on Computational Science and Its Applications (ICCSA 2012)*, Salvador de Bahia, Brazil, 2012, pp. 202–216.

[12] K. Hodnigg and R. T. Mittermeir, "Metrics-Based Spreadsheet Visualization - Support for Focused Maintenance," in *Proceedings of the 9th Annual Spreadsheet Risks Conference (EuSpRIG '08)*, London, United Kingdom, 2008, pp. 79–94.

[13] F. Hermans, "Improving spreadsheet test practices," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2013)*, Ontario, Canada, 2013, pp. 56–69.