Fragment-based spreadsheet debugging

Dietmar Jannach¹ · Thomas Schmitz² · Birgit Hofer³ ·
Konstantin Schekotihin¹ · Patrick Koch¹ · Franz Wotawa³

Received: 27 October 2017 / Accepted: 10 December 2018 / Published online: 22 December 2018
© The Author(s) 2018

Abstract

Faults in spreadsheets can represent a major risk for businesses. To minimize such risks, various automated testing and debugging approaches for spreadsheets were proposed. In such approaches, often one main assumption is that the spreadsheet developer is able to indicate if the outcomes of certain calculations correspond to the intended values. This, however, might require that the user performs calculations manually, a process which can easily become tedious and error-prone for more complex spreadsheets. In this work, we propose an interactive spreadsheet algorithmic debugging method, which is based on partitioning the spreadsheet into fragments. Test cases can then be automatically or manually created for each of these smaller fragments, whose correctness or faultiness can be easier assessed by users than test cases that cover the entire spreadsheet. The annotated test cases are then fed into an algorithmic debugging technique, which returns a set of formulas that could have caused any observed failures, i.e., discrepancies between the expected and computed calculation outcomes. Simulation experiments demonstrate that the suggested decomposition approach can speed up the algorithmic debugging process and significantly reduce the number of fault candidates returned by the algorithm. An additional laboratory study shows that fragmenting a spreadsheet with our method furthermore reduces the time needed by users for creating test cases for a spreadsheet.

Keywords Spreadsheets · Algorithmic testing and debugging · Artificial intelligence · Model-based diagnosis

---

The work is based on and significantly extends our previous work presented in Schmitz et al. (2016, 2017).

✉ Dietmar Jannach
dietmar.jannach@aau.at

Extended author information available on the last page of the article

1 Introduction

Spreadsheets are widely used in organizations, e.g., for automating daily tasks, for financial accounting and controlling, and also for preparing major business decisions (Scaffidi et al. 2005). However, even though spreadsheets are used for potentially critical tasks, they are often not subject to a systematic quality assurance process. As a consequence, faults in spreadsheets have caused substantial losses for companies in the past.¹

Different approaches exist to minimize the risk that spreadsheets contain faults. One predominant approach in the literature is to aim at better *tool support* for users for spreadsheet *testing* and *algorithmic debugging* (Jannach et al. 2014), and studies like (Roy et al. 2017) in fact suggest that users would appreciate if better testing support would be available in spreadsheet environments. Many of the approaches proposed in the literature, however, at some stage require the user to define test cases that specify the expected calculation output values when given certain inputs (Abreu et al. 2015). Or, they at least ask users to indicate if the computed output values are correct or faulty, i.e., if the test case revealed a failure (Hofer et al. 2013; Reichwein et al. 1999; Ruthruff et al. 2005). Overall, while the general task itself might be intuitive for users, defining test cases for larger spreadsheets can easily become a tedious and error-prone task.

In this paper, we therefore propose a method and corresponding tool that combines the general idea of using test cases for algorithmic debugging with a novel approach that decomposes complex spreadsheets into smaller, more comprehensible pieces, which we call fragments. The underlying intuition is that users will find it easier to specify the expected outcomes of calculation chains when only a part of the spreadsheet has to be considered.

Testing individual *units* of a program in isolation is common practice in today's software development processes. Spreadsheets, however, not only have a very distinct structure, they are also often developed by users that have limited expertise in modern software engineering. We therefore propose an approach where a given spreadsheet is *automatically* decomposed by the system according to a set of heuristics. The spreadsheet developers can then provide test cases for these, usually much smaller, sets of calculations. Using our tool, they can also let the system automatically *generate test cases*. The resulting test cases are then used as inputs to the model-based and constraint-based spreadsheet debugging approach and tool proposed in Jannach and Engler (2010) and Jannach and Schmitz (2016).

We have experimentally validated our proposal in three different dimensions. First, we made simulation experiments indicating that the computation times of the debugging algorithm can be lowered when fragments are used as inputs. Moreover, in this evaluation we found that the application of fragments substantially decreased the number of fault candidates that were returned by the debugging algorithm. Second, a set of further simulation experiments showed that the fragmentation approach in combination with automatically generated test cases reduces the required effort for users

¹ https://urldefense.proofpoint.com/v2/url?u=http-3A__www.eusprig.org_horror-2Dstories.htm&d=DwIGaQ&c=vh6FgFnduejNhPPD0fl_yRaSfZy8CWbWnIf4XJhSxq8&r=cijxKIUfIjh6xB35XSxKelnSNfz2185wGO_qFr-DFH8&m=6Klepf_SfBcuNRlcQfxCWK6Kvi4FQQVulh5rCecVOs4&s=v1gScmI58oDlqqK9KRp3mowozw2cY5o60yS5artofw&e=

in terms of the numbers of formulas they have to inspect when trying to locate the faults. Third, we conducted a laboratory study, where the participants had to specific test cases with and without fragmentation support. The experiment showed that the proposed fragmentation approach increased the users' efficiency when providing test cases for the spreadsheet. Overall, the proposed decomposition method therefore not only makes it easier for users to specify test cases, these smaller test cases also make the debugging algorithm more effective.

The paper is structured as follows. We provide an introductory example in the next section that shows how fragments can help to reduce the complexity of providing test cases. In Sect. 3, we define our fragmentation approach more formally and present an algorithm to automatically find a suitable fragmentation. How these fragments and the corresponding test cases are finally integrated into an interactive algorithmic spreadsheet debugging technique is discussed in detail in Sect. 4. The results of the different empirical analyses are presented in Sect. 5. Sections 6 and 7 discuss related works and provide an outlook on future developments.

2 Motivating example

We use the example spreadsheet shown in Fig. 1 to illustrate our approach. The spreadsheet calculates the profit of a company that produces and sells different products A to H. The inputs of the spreadsheet are the production costs, the sales prices, and the units sold per month for each of the eight products (rows 7–14). In these rows, for each product the number of sold units, the revenue, and the overall production costs are calculated in the columns K–M. The overall figures can be found in cells L17–L20. Here, the totals per product are added up and the overall profit for the current year is determined.

Let us assume that a user wants to validate the correctness of the spreadsheet. One way of doing this is to check if the *output cells* of the overall figures, i.e., those cells

	A	B	C	D	E	F	G	H	I	J	K	L	M
1			Company calculation sheet										
2													
3													
4													
5	Product information			Units sold per month							Totals per product		
6	Product	Prod.cost/pc	Sale price/pc	Jan	Feb	Mar	Apr	May			Units sold	Revenue	Prod. cost
7	A	3	4	2	2	2	2	2			10	40	30
8	B	3	4	2	2	2	2	2			10	40	30
9	C	3	4	2	2	2	2	2			10	40	30
10	D	3	4	2	2	2	2	2			10	40	30
11	E	3	4	2	2	2	2	2			10	40	30
12	F	3	4	2	2	2	2	2			10	40	30
13	G	3	4	2	2	2	2	2			10	40	30
14	H	3	4	2	2	2	2	2			10	40	30
15													
16											Overall figures		
17											Units sold	80	
18											Revenue	320	
19											Prod. costs	210	
20											Profit	110	

Fig. 1 Example spreadsheet for profit calculation (Jannach and Schmitz 2016)

Fig. 2 A fragment that collapses cells with identical formulas. Green cells are input cells, yellow cells are intermediate calculations, and orange cells are the fragment's output cells (Color figure online)

that are not referenced by other cells, contain the expected values given the current inputs. These inputs and the corresponding outputs can be considered as a test case and, as mentioned above, several existing spreadsheet debugging approaches rely on such test cases.² However, even for this example spreadsheet with medium complexity, deciding, for example, if the value in cell L20 (the overall profit) is correct, is not trivial.

The general idea of our decomposition-based approach is therefore to split a given spreadsheet into smaller parts and to let the user specify test cases for these smaller fragments. Since such fragments contain fewer formulas and usually have shorter calculation chains, determining if the calculated outputs are correct should, as a result, be easier for the user. In addition to splitting up the spreadsheet into fragments, we propose to further simplify the structure of the fragments by “collapsing” cells that contain identical formulas, an approach that was proposed earlier also in Burnett et al. (2002). In the spreadsheet from Fig. 1, for example the formulas in cells K7–K14 all contain the same formula, but are applied to different rows of the input data. Usually, the contents of such formulas are copied from one cell (e.g., from K7 in that case). Since the formulas are structurally identical, it is therefore sufficient to provide a test case for one representative formula (e.g., K7), as in Burnett et al. (2002), because any fault introduced in the original cell K7 will also be found in the copied cells.

A possible fragmentation of the example spreadsheet into two parts, as created by our software tool, is shown in Figs. 2 and 3. The first fragment (shown on the right-hand side of Fig. 2) covers the calculations of the totals for the different products. The formula cells of the fragment are colored in yellow (indicating intermediate calculations within the fragment) and orange (indicating output cells of the fragment). The inputs used by the fragment, which are not part of the fragment itself, are highlighted in green color. In order to help the user to focus on the current fragment, the rest of the spreadsheet is grayed out. In the fragment shown in Fig. 2, the calculations are the same for each of the products, allowing the tool to collapse the cells accordingly.

² Alternatively, a user could also inspect each formula for correctness, independent of any test case. Such an approach however does not scale well to spreadsheets with many formulas.

Fig. 3 A fragment containing neighboring cells. Green cells are input cells, yellow cells are intermediate calculations, and orange cells are the fragment's output cells (Color figure online)

Hence, the user only has to specify a test case for the representative, product A. The identical formulas for products B to H are shown in lighter colors to indicate this. Overall, to check if a test case leads to the expected outcomes or reveals a *failure*, the user only has to check if the fragment's three cells K7, L7, and M7 contain the expected values given the inputs of the same row in cells B7–I7.

The second fragment of this spreadsheet is shown in Fig. 3 and covers the calculations of the overall figures in cells L17–L20. As these calculations use the totals per product as their input values, the cells from K7 to M14 serve as input values for this fragment and are therefore now colored in green. Although these cells in the original spreadsheet comprise formulas, users of our tool can type in values for these cells when testing this fragment in order to check the outcomes of the calculations of the fragment.³

Overall, with a fragmentation like this, not only the effort for the user to specify test cases is reduced, also the calculation chains become shorter. Specifically, given this fragmentation, the user only has to check the results of at most 7 formulas, with the longest calculation chain being of length 2. Without the fragmentation, up to 28 formulas would have to be checked, and the longest calculation chain would have contained four formulas. Other ways of fragmenting the given spreadsheet are, of course, possible. How to find “good” fragmentations will be discussed later in this paper. Generally, the approach presented in this paper also supports the manual definition of fragments, as well as the specification of test cases for the entire spreadsheet if this seems more useful to the developer.

3 An automated fragmentation approach

In this section, we describe an algorithmic approach to decompose a given spreadsheet into fragments in an automated way. We first define the problem more formally, and

³ The formulas of the original spreadsheet are, of course, left unchanged.

then present an evolutionary algorithm that constructs the fragments in a heuristic bottom-up approach.

Definition 1 (*Fragment and fragmentation*) Let S be a spreadsheet (a set of cells) and $formulaCells : S \mapsto S_{formulaCells}$ be a function that maps a set of cells S to its subset $S_{formulaCells}$ 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 $formulaCells(S) = \bigcup_{f_i \in F} f_i$.

A fragmentation of a spreadsheet therefore is a set of fragments that contain formula cells. If all formula cells of a spreadsheet are part of at least one fragment of the fragmentation we call the fragmentation to be covering. Normally, we are interested only in covering fragmentations so that a developer can use the fragmentation to provide test cases for all formulas of a spreadsheet. Definition 1 in general allows fragments to be overlapping, i.e., one formula cell can be part of more than one fragment. However, the automatic decomposition approach proposed in this paper creates fragmentations that are not overlapping in order to keep the size of the fragments small.

Example 1 For the example spreadsheet shown in Fig. 1, the fragmentation $F = \{f_1, f_2\}$ with $f_1 = \{K7, L7, M7, K8, \dots, M14\}$, $f_2 = \{L17, L18, L19, L20\}$ shown in Figs. 2 and 3 is covering, because all formula cells of the spreadsheet are part of a fragment of F .

Our automatic approach to generate a fragmentation has two phases:

1. In the first phase, we identify formulas of which we assume that they *semantically* do the same calculations. As described above, the idea is to *collapse* them in a way that the user only has to specify a single test case that is then applicable for all semantically-equivalent formulas.
2. In the second phase, we explore different ways of merging the fragments so that the resulting set of fragments *optimizes* a fitness function. This function combines heuristics assessing the “goodness” of a fragmentation, e.g., in terms of the number of fragments or the complexity of each fragment.

3.1 Collapsing semantically equivalent calculations

Determining in an automated way and with certainty that two formulas were copied and are semantically equivalent is, in the general case, not possible without having access to “external” information about the spreadsheet. In our approach, we therefore rely on heuristics as indicators that two formulas are in fact semantically equivalent and can be collapsed.

Generally, formulas can only be semantically equivalent if they perform the same calculations. Therefore, we only consider formulas as *candidates* to be semantically equivalent if they are “copy-equivalent”. In Mittermeir and Clermont (2002), the term copy-equivalence used to denote cells with identical formulas. To find such copy-equivalent formulas, we compare the formula definitions in R1C1 notation, which is a relative reference notation supported by MS Excel and other tools.

Definition 2 (*Copy-equivalence*) Two formulas are copy-equivalent if they are identical in R1C1 notation. We use the notation $\text{copy-equivalent}(c, c')$ to express the copy-equivalence of two cells c and c' of a given spreadsheet.

However, since two formulas could be also coincidentally copy-equivalent, we additionally require their cells to be in the same row or column. We use this heuristic because we observed that in many real-world spreadsheets users mostly copy formulas only to other cells in the same row or column. In some cases, however, users first copy formulas to other cells in the same row and then also copy one or all of these formulas to additional cells in the same column. To model this relaxed requirement, we define the concept of “column-row-related” fragments in addition to copy-equivalence. These are fragments where each cell shares the same row or column with at least one other cell of the fragment. To formally define this requirement, we use a connected graph with the property that two nodes of the graph are connected if and only if the cells represented by the nodes share the same row or column.

Definition 3 (*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.

Example 2 In the example spreadsheet shown in Fig. 1, the fragment $\{K7, K8, L8, L9, M9\}$ is column-row-related, as a connected graph exists for these cells such that edges connect two nodes if their cells share the same row or column. The fragment $\{K7, K8, L9, M9\}$ is, however, not column-row-related as no such graph can be constructed. The reason is that while K7 and K8 share the same column and while L9 and M9 share the same row, neither K7 nor K8 share the same row or column with any of L9 and M9.

Based on the concepts of copy-equivalence and column-row-relatedness, we can now define what we call a *base fragment*. Intuitively, a base fragment, which is later on possibly merged with other base fragments in the automated fragmentation process, is either a single formula or represents a set of semantically-equivalent cells. We consider formulas to be semantically-equivalent if they are copy-equivalent and column-row-related.

Definition 4 (*Base fragment*) A column-row-related fragment f_b is called a *base fragment*, if the condition $\forall c, c' \in f_b : \text{copy-equivalent}(c, c')$ holds for f_b , but does not hold for any column-row-related fragment $f'_b \supset f_b$.

A base fragment according to the definition is the largest possible fragment that is (i) column-row-related and (ii) where all cells are copy-equivalent. For each base fragment, a *representative cell* is defined as follows.

Definition 5 (*Representative cell*) The left-most cell c in the top-most row of a base fragment f_b is called the *representative cell* of f_b . The function $\text{rpc}(f_b) : f_b \mapsto c$, where c satisfies the condition $\forall c' \in (f_b \setminus \{c\}) : y(c) < y(c') \vee (y(c) = y(c') \wedge x(c) < x(c'))$, returns the representative cell of f_b .

The representative cell of a base fragment will be used for the specification of the test cases. In case the base fragment contains copy-equivalent cells, these test cases can then be applied to all copies of the representative cell.

Example 3 For the example spreadsheet shown in Fig. 1, the following seven base fragments exist: $f_1 = \{K7, K8, \dots, K14\}$, $f_2 = \{L7, L8, \dots, L14\}$, $f_3 = \{M7, M8, \dots, M14\}$, $f_4 = \{L17\}$, $f_5 = \{L18\}$, $f_6 = \{L19\}$, and $f_7 = \{L20\}$. The representative cells for these fragments are given as $rpc(f_1) = \{K7\}$, $rpc(f_2) = \{L7\}$, $rpc(f_3) = \{M7\}$, $rpc(f_4) = \{L17\}$, $rpc(f_5) = \{L18\}$, $rpc(f_6) = \{L19\}$, and $rpc(f_7) = \{L20\}$.

3.2 Merging fragments

The set of base fragments in principle already represents a possible fragmentation of a given spreadsheet. However, in case that there are no copy-equivalent cells in the spreadsheet, each cell with a formula would represent a base fragment. This would, however, not be very helpful in the test case specification process, as the fragmentation is too fine-grained and too many fragments exist for which test cases have to be specified. Our goal is therefore to merge two or more base fragments that are, in the best case, logically connected. At the same time, the resulting merged fragments should not be too complex (e.g., in terms of the number of formulas), given that our main goal is to reduce the cognitive load for the user when specifying test cases.

Since there is no standard way to calculate the complexity of a spreadsheet fragment, we designed a heuristic function based on considerations from the academic literature on spreadsheet smells (Cunha et al. 2012; Hermans et al. 2012) and spreadsheet complexity measures (Hodnigg and Mittermeir 2008). In the following, we describe how we assess the complexity of a fragment based on heuristics.⁴

Our heuristic generally only considers the representative cells of the fragments as inputs, as the user can basically ignore all copy-equivalent calculations. We use the following function to obtain all representative cells of a fragment that was created by merging two or more base fragments.

Definition 6 (*Representatives*) Let $f = \{f_{b1}, \dots, f_{bn}\}$ be a fragment consisting of n base fragments. The function $Rps(f)$ returns the set of representative cells of the base fragments:

$$Rps(f) := \bigcup_{f_{bi} \in f} rpc(f_{bi}).$$

To calculate the complexity of an individual fragment, we use a heuristic that considers four different components: the number of inputs, the number of outputs, the area covered by the fragment, and the complexity of its formulas.

$$H_{in}(f) := |input(Rps(f))| \quad (1)$$

⁴ Alternative heuristics can be easily embedded into our method.

$$H_{out}(f) := |\text{output}(Rps(f))| \quad (2)$$

$$H_{area}(f) := (\max_x(Rps(f)) - \min_x(Rps(f)) + 1) \\ * (\max_y(Rps(f)) - \min_y(Rps(f)) + 1) \quad (3)$$

$$H_{formulas}(f) := \sum_{c \in Rps(f)} \text{formulaComplexity}(c) \quad (4)$$

In the listed equations, 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 counting the number of conditionals and cell references in it. In case of a formula that is referring to cell ranges, this function only counts the number of written references, e.g., the reference A1:A10 would result in a complexity value of 2, since larger ranges do not cause a formula to be more complex.

Additional heuristics could be used here as well to determine the complexity of a fragment, for example, fault probabilities or the length of the calculation chains. However, as the results of an empirical evaluation will show, already these comparably simple heuristics led to fragmentations that were helpful for users when specifying test cases.

Example 4 When using the specific formulas of our example spreadsheet shown in Fig. 4, our heuristics lead to the following values for the fragmentation $F = \{f_1 = \{K7, L7, M7, K8, \dots, M14\}, f_2 = \{L17, L18, L19, L20\}\}$:

Fragment	f_1	f_2
H_{in}	7	24
H_{out}	2	2
H_{area}	3	4
$H_{formulas}$	6	8

The overall complexity of a fragment is finally calculated by using a weighted sum of the four components. Because the heuristics differ both in their importance and in the scale of their values, we assign a weight to each heuristic. The weights generally depend on the used heuristics and can be set based on expert knowledge.

$$\text{fragComp}(f) := \sum_{i=1}^{|H|} H_i(f) * w_i \quad (5)$$

Example 5 Given the weight vector $w = (2, 3, 4, 5)$, the complexities for the fragments of Example 4 are $\text{fragComp}(f_1) = 7 * 2 + 2 * 3 + 3 * 4 + 6 * 5 = 62$ and $\text{fragComp}(f_2) = 24 * 2 + 2 * 3 + 4 * 4 + 8 * 5 = 110$.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1			Company calculation sheet										
2													
3													
4													
5	Product information			Units sold per month					Totals per product and year				
6	Product	Prod.cost/pc	Sale price/pc	Jan	Feb	Mar	Apr	May	Units sold	Revenue	Prod. cost		
7	A	3	4	2	2	2	2	2	=SUM(E7:I7)	=K7*C7	=K7*B7		
8	B	3	4	2	2	2	2	2	=SUM(E8:I8)	=K8*C8	=K8*B8		
9	C	3	4	2	2	2	2	2	=SUM(E9:I9)	=K9*C9	=K9*B9		
10	D	3	4	2	2	2	2	2	=SUM(E10:I10)	=K10*C10	=K10*B10		
11	E	3	4	2	2	2	2	2	=SUM(E11:I11)	=K11*C11	=K11*B11		
12	F	3	4	2	2	2	2	2	=SUM(E12:I12)	=K12*C12	=K12*B12		
13	G	3	4	2	2	2	2	2	=SUM(E13:I13)	=K13*C13	=K13*B13		
14	H	3	4	2	2	2	2	2	=SUM(E14:I14)	=K14*C14	=K14*B14		
15													
16									Overall figures				
17									Units sold	=SUM(K7:K14)			
18									Revenue	=SUM(L7:L14)			
19									Prod. costs	=SUM(M7:M13)			
20									Profit	=L18-L19			

Fig. 4 The formulas of the example spreadsheet

The complexities of the individual fragments can then be used to assess the overall complexity of a spreadsheet fragmentation. In order to find the best possible fragmentation, i.e., the least complex one, we use an evolutionary algorithm that optimizes the fragmentation based on the complexity of the individual fragments. Evolutionary algorithms, in general, simulate the process of evolution and optimize a given problem over multiple generations. In each generation, only the best candidates survive and form the population from which the new generation is created by crossing over and randomly mutating the survivors of the last generation. The survival of candidates in such algorithms is determined by a fitness function. In our case, the fitness of a fragmentation F is based on the complexities of its constituent fragments and is computed as follows.

$$fitness(F) := - \left(\sum_{f \in F} fragComp(f) \right) - |F| * w_F - \sigma(F) * w_\sigma \quad (6)$$

Because less complex fragmentations are supposed to be better, the sum of the fragment complexities is negated. In addition, our fitness measure considers (a) the number of fragments in the fragmentation and (b) how strongly the individual fragments vary in terms of their complexity. This second aspect is introduced to penalize fragmentations which consist, e.g., of one very complex and large fragment and many smaller and very simple fragments. The strength of the penalties is in both cases determined by some weight parameter that we set manually in our experiments. In the formula, w_F is the importance factor for the number of fragments, and w_σ the penalty factor when the complexities of the fragments vary a lot, expressed in the standard deviation. The standard deviation of the fragment complexity values is computed as follows.

$$\sigma(F) := \sqrt{\frac{\sum_{f \in F} \left(\text{fragComp}(f) - \frac{\sum_{f \in F} \text{fragComp}(f)}{|F|} \right)^2}{|F|}} \quad (7)$$

Example 6 Using the weight factors $w_F = 1$ and $w_\sigma = 2$, the overall fitness of the fragmentation F of Example 4 is $\text{fitness}(F) = -(62 + 110) - 2 * 1 - \sqrt{\frac{(62-86)^2 + (110-86)^2}{2}} * 2 = -222$.

The resulting evolutionary procedure to find optimal or “close-to-optimal” fragmentations is listed in Algorithm 1. The algorithm accepts the spreadsheet and the weight parameters as input and returns one *covering* fragmentation as output. In addition, the usual parameters for evolutionary algorithms⁵ have to be specified.

Algorithm 1: Evolutionary Algorithm for Fragment Generation

Input: A spreadsheet S , population size p , survival rate s , number of generations without any improvement g , heuristic weights w , w_F , and w_σ

Output: A covering fragmentation F

```

1  $B \leftarrow \text{createBaseFragments}(S)$ ;
2  $P \leftarrow \text{createInitialPopulation}(B, p)$ ;
3  $\text{count} \leftarrow 0$ ;
4 while  $\neg \text{shouldStop}(P, \text{count}, g)$  do
5    $P \leftarrow \text{selectFittestIndividuals}(P, s, w, w_F, w_\sigma)$ ;
6    $P \leftarrow P \cup \text{getNewGeneration}(P, p - |P|)$ ;
7    $\text{count} \leftarrow \text{count} + 1$ ;
8  $F \leftarrow \text{selectFittestIndividual}(P, w, w_F, w_\sigma)$ ;
9 return  $F$ ;
```

The algorithm first creates all base fragments for the given spreadsheet according to Definition 4. At the beginning of the evolutionary process, an initial population consisting of p fragmentations is generated by randomly merging the base fragments. In the while loop, the algorithm selects the $s * p$ fittest individuals of P and removes all weaker fragmentations. The remaining fragmentations are recombined by randomly merging fragments or by splitting them into their base fragments. This way, newly obtained individuals are added to the population to be considered in the next step, until the population size again reaches its defined limit p . The while loop continues until some stopping criterion is met. In our implementation, the while loop stops when the fittest individual of P could not be improved during the last g generations.⁶ In this case, the fittest individual is returned as the optimal or close-to-optimal fragmentation.

For our experiments, we used the following weight factors: $w = \{0.1, 1, 1, 1.5\}$, $w_F = 0.05$, and $w_\sigma = 0.1$. The general algorithm parameters were as follows:

⁵ Specifically, the parameters are the population size p , the survival rate s , and the number of generations without improvement g . For an introduction to evolutionary algorithms, see, e.g., (Russell and Norvig 2009).

⁶ Technically, the procedure *shouldStop* internally stores the best fitness value so far and the number of calls without an improvement, e.g., in global variables.

$p = 100$, $g = 25$, $s = 0.9$. The weight factors were, as mentioned above, determined in a manual process, where we systematically varied the parameter values and decided based on the outcomes of the fragmentation process for different spreadsheets if the resulting fragments are meaningful. As our results will show, the resulting fragmentations were helpful for the study participants. Further optimizations are possible, but are left for future work.

4 Using fragments for algorithmic debugging

In Sect. 3, we have shown how we can automatically decompose a spreadsheet into smaller chunks for which we assume that it is easier for users to specify test cases. Such test cases play a central role in the integrated fragment-based spreadsheet debugging approach which we will present in more detail in this section. In the following, we will first briefly summarize the background of the central algorithmic debugging approach as proposed in Jannach and Engler (2010) and Jannach and Schmitz (2016). We will then discuss how fragment-based test cases can help to increase its effectiveness. Finally, we describe the overall process when using our approach, starting from automated fragmentation, over test case specification, to interactive fault localization.

4.1 Constraint-based spreadsheet debugging

The debugging method used in our work is based on the principles of Model-Based Diagnosis (MBD) (Reiter 1987). MBD is a general and domain-independent approach to fault localization, which is based on a systematic way of testing different hypotheses why a system under observation does not behave as expected. MBD techniques were historically developed in the context of the localization of faults in electronic circuits. The underlying assumption of such approaches is that a “model” of the system is given, i.e., that we know how the system, which usually consists of connected components, behaves when there is no fault. Test cases (or “observations” using MBD terminology) are then used to compare the expected outputs of a system to the observed outputs. Specific diagnosis algorithms can then be used to generate and test different hypotheses about which parts of the system are potentially broken.

In Jannach and Engler (2010), Jannach and Schmitz (2016), we have shown how the MBD principles can be applied to the spreadsheet debugging problem. Similar to electronic circuits, a spreadsheet has a set of inputs and outputs. In addition, a formal description of the behavior of the spreadsheet is given, since the outcomes of the formulas are computed according to the defined semantics of the operators and built-in functions of the spreadsheet environment. Given certain input values and unexpected output values, we can then test—using standard diagnosis algorithms—which formulas of the spreadsheet are potentially contributing to outputs that deviate from the expected values that are specified in the test cases. Technically, to test these hypotheses, the spreadsheet is translated into a constraint program as demonstrated in Jannach and Engler (2010), Jannach and Schmitz (2016).

In Reiter (1987), Reiter developed a formal characterization of the model-based diagnosis problem using a logic-based framework. Here, we will only briefly discuss how these ideas translate to the spreadsheet debugging problem. In Reiter’s work, a diagnosable system is defined as follows.

Definition 7 (*Diagnosable system*) A diagnosable system is a pair $(SD, COMPS)$ where SD is a system description (a set of logical sentences) and $COMPS$ represents the system’s components (a finite set of constants).

In the context of spreadsheets, the set $COMPS$ corresponds to the formula cells of a given spreadsheet (the system under observation) and SD describes the logic of the formulas. The expected behavior of a component (a formula) is expressed by formally describing the component’s output when it is behaving normally. In the literature, when a logic language is used to formally describe the correct behavior, a special unary “abnormal” predicate $AB(\cdot)$ is traditionally used for that purpose. The statement “ $AB(K7) \vee K7 = SUM(E7, \dots, I7)$ ” would therefore mean that the value of $K7$ is faulty (behaving abnormally) or equals the sum of the values in cells $E7$ – $I7$.

Example 7 For our running example shown in Fig. 1, we have $SD = \{AB(K7) \vee K7 = SUM(E7, \dots, I7), AB(L7) \vee L7 = K7 * C7, \dots, AB(L20) \vee L20 = L18 - L19\}$ and $COMPS = \{K7, \dots, M14, L17, L18, L19, L20\}$.

A diagnosis problem arises when the system’s outputs for a set of inputs (all of them expressed in logical sentences in Reiter’s framework) are inconsistent with the expected outputs. In software engineering terminology, this corresponds to the situation when a test case elicits a *failure*.

A pair of inputs and expected outputs is called “observations” in the framework and corresponds to a test case in our setting. While Reiter’s formulation was based on using a single test case, (Felfernig et al. 2004) showed how multiple test cases, stored in a set TC , can be embedded into the framework.

Example 8 In our example, the set TC contains a single test case with the input values shown in Fig. 1 and the expected values for the output cells, i.e., $TC = \{B7 = 3, C7 = 4, \dots, I14 = 2, L20 = 80\}$.

A diagnosis in Reiter’s framework is then a subset of the system’s components which, if assumed to behave in an unexpected, faulty way, explain (are consistent with) the observed outputs for all test cases in TC .

Definition 8 (*Diagnosis*) Given a diagnosis problem $(SD, COMPS, TC)$, a diagnosis is a minimal set $\Delta \subseteq COMPS$ such that $\forall T \in TC : SD \cup T \cup \{AB(c) \mid c \in \Delta\} \cup \{\neg AB(c) \mid c \in COMPS \setminus \Delta\}$ is consistent.

Translated to the spreadsheet debugging problem, a diagnosis is a set of formulas that are potentially faulty, i.e., if we make no assumptions about their outputs for a given test case, the expected outputs are possible. A detailed analysis of the effectiveness of applying MBD principles to the spreadsheet debugging problem can be found in Jannach and Schmitz (2016).

To find all diagnoses, Reiter proposes the Hitting Set Tree (Reiter 1987) algorithm, which systematically tests all possible causes for the unexpected behavior. In Reiter's logic-based framework, a general *theorem prover* is used to compute the expected outcomes given some inputs and fault assumptions. In the context of spreadsheet debugging, we use a constraint solver similarly to Jannach and Schmitz (2016).

4.2 Constraint-based spreadsheet debugging with fragments

In this work, we hypothesize that a fragment-based approach can help to improve the described constraint-based spreadsheet testing and debugging approach in the following ways. We will discuss the results of the corresponding empirical analyses later in Sect. 5.

1. *Reducing user effort* One main hypothesis is that users find it easier to specify test cases for smaller fragments instead of the entire spreadsheet. As a consequence, the entire debugging process, which is based on such test cases, is assumed to be more efficient.
2. *Improving computational efficiency* The use of fragments can also be helpful to speed up the diagnostic reasoning process in two ways.
  - (a) First, when a spreadsheet is decomposed into several parts, the principles of what is called *hierarchical diagnosis* can be applied (Siddiqi and Huang 2007). The idea of such approaches is to start the diagnostic reasoning process at a more coarse-grained level, i.e., not on the level of individual formulas but on the level of fragments. The outcome of the computations at the upper level is then a set of fragments that potentially contain a fault. Given these high-level computations, which can be usually done efficiently, one can iteratively refine the diagnoses until one reaches the level of individual formulas; see (Felfernig et al. 2001) for such an approach of iterative focusing.
  - (b) Second, when using fragment-based test cases, we may assume that some of the computations will be faster, because more fine-grained information is available about the potential causes of a failure, i.e., the search process can be more focused.
3. *Improving the effectiveness of algorithmic debugging* When test cases are defined for smaller portions of the spreadsheets, which is the case for fragment-based test cases, they are potentially more helpful to isolate the true cause of the unexpected behavior than when a test case spans an entire spreadsheet. In the context of our constraint-based debugging approach, we therefore expect that using test cases based on fragments leads to fewer diagnosis candidates.

4.3 Summary of overall workflow and tool functionality

In this section, we summarize the overall workflow of testing and debugging spreadsheets using fragments within our constraint-based debugging approach. The entire workflow is implemented within an add-in to MS Excel called EXQUISITE. The EXQUISITE add-in is designed in a way that users can remain within their usual spread-

Fig. 5 Ribbon with testing and debugging functionality of EXQUISITE

sheet environment and invoke the corresponding testing and debugging functionality on demand. A part of the “ribbon” of the add-in to MS Excel is shown in Fig. 5. A brief summary of the tool’s functionality can also be found in Schmitz and Jannach (2017).

During or after the design of the spreadsheet, a user of EXQUISITE can perform the following steps.

1. *Generate and adapt fragments* This functionality automatically decomposes the spreadsheet using the evolutionary algorithm described in Sect. 3. After the fragmentation is computed, the different fragments of the spreadsheets are visually highlighted as shown in Fig. 2. Alternatively, the user can define the fragmentation in a manual process and/or correct the system’s proposal, e.g., in cases where our heuristics missed a number of semantically-equivalent cells.
2. *Rank fragments by fault probability* Once the fragmentation is defined, users can specify test cases for each of the fragments and the system can automatically rank the fragments according to the probability that the fragment will contain at least one faulty formula. The framework supports different methods to estimate fault probabilities for formulas. In the default implementation, we use the complexity of the fragment (as described in Sect. 3) as an indicator for the probability that a fragment contains a fault.
3. *Generate test cases for fragments* To support the user in the process of writing test cases, our add-in supports the automated generation of test cases. Different techniques from the literature, like Abraham and Erwig’s *AutoTest* approach (Abraham and Erwig 2006), can in principle be used to generate different test cases that, e.g., cover all calculation paths of a spreadsheet. In our implementation, we use a simple method that generates individual test cases randomly using defined ranges for input values, e.g., ranging from 2 to 10.⁷
4. *Label test cases* After the inputs for a test case have been generated or manually specified, we ask the user to check the correctness of the calculation outcomes. Again, due to the often limited size of the fragments, we assume that checking the correctness is easier for individual fragments than for the entire spreadsheet. In cases where a test cases elicits a failure, i.e., the calculation outcome is wrong, the system asks the user to provide the correct value.⁸ If at least one test case is defined for a fragment and this test case revealed no failure, we mark the fragment as “tested”. The user can, however, still add additional test cases. The system then again ranks the remaining untested fragments according to their fault probability and encourages the user to specify test cases for these fragments as well.

⁷ Zeros and ones are not used, because they are neutral elements for the multiplication and addition operators.

⁸ Technically, it is also possible for the user to only specify a range of values.

5. *Initiate algorithmic debugging process* If a test case is labeled as leading to a failure, the user can start the debugging algorithm. The system will then apply the MBD-technique described in Sect. 4.1 and return a list of diagnosis candidates which is displayed to the user. The user can then select individual diagnoses in the add-in and inspect the involved and potentially faulty formulas.⁹

5 Evaluation

We evaluated our fragment-based approach to spreadsheet testing and debugging in three different ways.

1. We ran simulation experiments to investigate if the use of fragments can help to increase the efficiency of algorithmic debugging, i.e., reduce the computation times required by the constraint-based debugging technique (Sect. 5.1).
2. We conducted additional simulation experiments to assess to what extent the automated fragmentation process and the generation of test cases can help to increase the effectiveness. Our approach is effective if it reduces the effort for users in terms of how many formulas they have to inspect to locate a fault (Sect. 5.2).
3. We finally conducted a user study where the task of the participants was to specify test cases with and without the use of fragments, where one main goal was to assess if fragments make this process indeed easier for users as hypothesized. This study also helped us to assess the general usefulness of the approach as perceived by end users (Sect. 5.3).

5.1 Study I: using fragments for hierarchical diagnosis

As described in Sect. 4.2, fragments can be used as part of a hierarchical diagnosis process. In such a setting, the fault localization process is first done on a coarse-grained level, and the output of the diagnostic reasoning process is a set of potentially faulty fragments instead of a set of formulas. Once the location of a potential fault is determined on that coarse-grained level, additional test cases can be used to precisely locate it within the affected fragments.

The expectation for this experiment is that—similar to past works on hierarchical diagnosis (Felfernig et al. 2001; Siddiqi and Huang 2007)—the diagnostic process is faster, due to the reduction of the search space caused by the smaller number of potentially faulty components to consider. While applying MBD on the coarse-grained information results in less precise initial information, we expect that the subsequent refinement of the diagnoses (using additional test cases) will help us to better narrow down the set of diagnosis candidates.

---

⁹ In cases where multiple diagnoses are returned, a “sequential” diagnosis approach, as described in Shchekotykhin et al. (2016), can be initiated as well. In sequential diagnosis approaches, users are asked for additional feedback on the correctness of intermediate values or formulas, which can then be used to iteratively narrow down the set of diagnoses until the true reason of the unexpected behavior is found.

Table 1 Characteristics of the 4 spreadsheets from (Jannach and Schmitz 2016) and the 21 tested spreadsheets from the Enron Errors Corpus

ID	Spreadsheet name	#Input cells	#Formulas	#Unique formulas	#Faults
A	Wage planning	69	63	25	1
B	Proteins	14	98	14	2
C	Sales forecast	224	143	4	2
D	Course planning	126	457	10	2
1	HPLN0601	20	6	3	2
2	Southpoint gas model	51	26	2	1
4	Tricia roome	241	199	22	1
5	July01invoice rev1	235	680	25	1
6	Plan-real-time trading	352	227	50	1
8	Sun Devil project request	120	38	14	1
9	ONEOKRECAP2001	6022	5156	41	1
10	SunDevil01CostSummary	8	14	7	1
11	Citrus Pass-through 12.01.00	52	120	20	2
12	Huber-existing & proposed deals 111,201	3110	2385	34	191
13	Agro case question 3—Jeff	19	8	8	3
18	CrackSpreadOptions	68	84	2	6
19	EPE_MAY_2002	1272	316	15	24
20	oct2301	825	586	69	1
21	SoCalIntCalc	419	345	47	1
23	Enron corp information request - Nov 2001	3549	887	64	1
25	Ogleth200103	85	209	12	28
26	Year 2001 commissioning summary 2-26-02	2805	6774	71	4
27	ENA_PGE_JUNE_1_2002	528	171	7	14
28	Southpoint gas model	51	26	3	7
30	Global hot list 1207	351	113	23	2

5.1.1 Experiment design

We used four real-world spreadsheets (named *A* to *D*) containing faults that were used previously for performance measurements in Jannach and Schmitz (2016). In addition to these four spreadsheets, we also evaluated our approach on 21 real-world spreadsheets from the Enron Errors Corpus (Schmitz and Jannach 2016). We compared the cell-based debugging algorithm proposed in Jannach and Schmitz (2016) with the fragment-based hierarchical variant described above in two dimensions: running times and the number of returned diagnoses. As the fragment-based diagnosis outputs are not as detailed as the cell-based ones, we proceeded as follows for the fragment-

based approach for a fair comparison. After computing the diagnoses at the fragment level, we automatically created *one* additional test case for each fragment that was part of a diagnosis as described above. Then, we restarted the diagnosis process with this additional test case on the cell level in order to precisely locate the fault. All experiments were run on a laptop computer with an Intel Core i7-4710MQ CPU running Windows 10. To factor out random effects, every simulation run was repeated 100 times.

The characteristics of the spreadsheets are shown in Table 1. The spreadsheets vary in their size and in their number of unique formulas. As it is common for spreadsheets, they often contain a number of copy-equivalent formulas. As a result, the number of unique formulas is much lower than the total number of formulas. Most spreadsheets contain exactly one fault. The *proteins* spreadsheet is an example that contains two unique faults; in several other cases, a faulty formula was copied by the user to other cells. Since our add-in to MS Excel has certain limitations, we had to slightly adapt some of the spreadsheets of the Enron corpus, e.g., by moving all cells to a single worksheet. The dependency structure and the complexity of the spreadsheets were, however, left unchanged.

5.1.2 Results

Table 2 shows the results of the fragmentation process. For all but two of these spreadsheets, the fragmentation process finished in less than one second, which indicates the applicability of the method under the narrow time constraints of interactive debugging. Even in the most complex case, where the optimal fragmentation involved 77 fragments and over 6000 collapsed cells, the process did not last longer than 3.5 s. The number of resulting fragments is usually much lower than the number of formula cells for each spreadsheet, mostly because many cells could be collapsed, thereby largely reducing the complexity of the spreadsheet. For spreadsheet 23 (*Enron corp information request - Nov 2001*), however, which contained 64 unique formulas, our fragmentation heuristic led to 67 fragments. In this particular case merging the base fragments of the spreadsheet led to lower complexity heuristics only in one case.

The results regarding the computation of diagnoses are shown in Table 3. Remember that the cell-based diagnosis method searched for faults at the cell level, whereas the fragment-based approach used the fragments as the smallest diagnosable parts. In this latter case, one additional test case was created for these fragments and the diagnosis was performed again on the cell level. Therefore, the last column (overall time) shows the sum of the times needed for the fragmentation process, running the diagnosis process on the fragment-level, and running it at the cell-level with the additional test cases.

Generally, the original cell-based algorithm needed between 1 ms and almost 12 h (as an extreme case) to compute the diagnoses. The number of diagnosis candidates ranged between 1 and over 2000. The following additional observations can be made:

1. Computing diagnoses at the more abstract fragment level as shown in the middle block of columns in Table 3 can be done in almost no time in many cases or at least

Table 2 Results of the fragmentation process

ID	Time (ms)	#Fragments	#Collapsed cells	#Merged cells
A	151	13	38	12
B	55	4	84	10
C	26	4	139	0
D	49	4	447	6
1	7	3	3	0
2	7	3	23	0
4	97	22	177	0
5	219	15	655	10
6	489	45	177	5
8	39	12	24	2
9	433	41	5112	3
10	12	5	7	2
11	87	18	99	3
12	623	11	2350	24
13	15	7	0	1
18	5	2	82	0
19	104	9	300	7
20	2972	56	516	14
21	923	30	297	18
23	766	67	819	1
25	34	8	197	4
26	3403	77	6691	7
27	18	8	163	0
28	6	4	22	0
30	90	25	88	0

much faster than diagnosing on the cell level. Moreover, the number of fragments that remain to be considered by the user or by the diagnostic process at the next level of detail can be more than halved in most cases, with many of the fragments not requiring any further consideration.

2. The results for the hierarchical approach (first fragment-based, then cell-based) are shown in the last block of columns in Table 3. In all cases, one additional test case for each possibly faulty fragment was, in fact, sufficient to pinpoint the exact cause of the problem (i.e., the number of diagnoses was one). The overall computation times for the three-phase process including the fragmentation process, fragment-based diagnosis, and cell-based diagnosis using the additional test cases could at the same time be significantly decreased for the non-trivial spreadsheets when compared to the original approach. Only for the most simple spreadsheets the overall computation time of our fragment-based approach exceeded the time of the cell-based approach. At the same time, however, this small overhead helped to

Table 3 Results of the fragment-based diagnosis procedure of Study I (see Sect. 5.1) for the 21 tested spreadsheets from the Enron Errors Corpus. Times are given in ms. For some spreadsheets the solver was not able to find a solution within 24 h because of the complexity induced by the test cases that covered the entire spreadsheet. This applied to both the cell-based as well as the fragment-based method

ID	Cell-based		Fragment-based		Cell-based using additional test cases		Overall time
	Time	#Diag	Time	#Diag	Time	#Diag	
1	1	1	1	1	1	1	8
2	10	24	1	2	0	1	8
4	25	2	4	2	2	1	102
5	Solver was not able to find a solution within 24 h						
6	96	37	17	6	3	1	508
8	710,794	28	570	5	0	1	610
9	46,204	47	2609	3	1424	1	4467
10	0	2	0	1	0	1	12
11	Solver was not able to find a solution within 24 h						
12	Solver was not able to find a solution within 24 h						
13	80	7	32	4	1	1	47
18	1	1	1	1	3	1	9
19	302	25	11	1	769	1	884
20	178	11	38	3	31	1	3041
21	52,966	20	631	5	6	1	1561
23	905	12	286	4	451	1	1503
25	Solver was not able to find a solution within 24 h						
26	Solver was not able to find a solution within 24 h						
27	14	5	3	1	59	1	80
28	9	24	0	1	1	1	8
30	44	25	5	3	3	1	99

narrow down the set of diagnoses to one element, i.e., the user, at least in theory,¹⁰ would only have to inspect the identified cells to locate the fault.

5.1.3 Summary and limitations

The experiments of Study I show that the fragmentation process itself is not particular time-consuming and can be done within the time constraints of interactive debugging settings. For most cases, the fragmentation approach proved to be highly beneficial in terms of the time that is required for the algorithmic debugging process. Furthermore, applying a hierarchical approach and using additionally generated test cases helped

¹⁰ This particular measurement assumes “perfect bug understanding”, i.e., that the user would always recognize the problem when inspecting a faulty formula. Works like (Parnin and Orso 2011) suggest that this assumption might not always hold in practice.

to focus the debugging process in such a way that only one diagnosis candidate was returned at the end.

With respect to the fragmentation outcomes, we observe that our general procedure was effective in the sense that it managed to collapse equivalent cells and to merge cells into a manageable set of fragments. A systematic analysis of the quality of the resulting fragmentation remains, however, to be done. A part of our future work is therefore to evaluate the subjective quality of the resulting fragmentations by end users and spreadsheet experts. Based on this feedback, our goal is to also further improve the parameters and heuristics of the evolutionary fragmentation algorithm.

A further general limitation of our work is that it is based on a limited number of spreadsheets so far. While the analyzed spreadsheets are based on real-world applications, further experiments are required to better understand how representative these cases are and whether there are special types of spreadsheets for which the proposed fragmentation process is not particularly effective.

5.2 Study II: reducing test case specification efforts

In this second series of experiments, our goal was to quantify to what extent the proposed fragmentation heuristics can reduce the required test effort for the user. Specifically, we investigate how many formulas a user actually has to inspect to label a test case as being correct or not. The underlying assumption is that, following the fragment-based approach, considering only a smaller fraction of the spreadsheet's formulas with test cases is in many cases sufficient to locate a fault with the algorithmic debugging approach.

5.2.1 Experiment design

To quantify this aspect, we used the spreadsheets of the Enron Errors Corpus, because this is the only corpus available with real-world spreadsheets and real faults.¹¹ We applied the following experimental procedure which simulates a possible strategy of a user who incrementally adds test cases for the individual fragments.

1. We applied our fragmentation algorithm to the spreadsheets as described in Sect. 3 and ranked the fragments in two alternative orders, as will be specified below.
2. We iterate through the fragments until our algorithmic debugging method returns a set of fault candidates or no untested fragments exist.
  - (a) Within the iteration, we automatically create a test case using the approach described in Sect. 4.3.
  - (b) If a test case fails, i.e., leads to outputs that are different from the expected ones¹², the algorithmic debugging approach is initiated to compute a set of diagnoses, and the iteration is ended.

¹¹ While the spreadsheets used in Jannach and Schmitz (2016) are also real-world cases, the faults in them were artificially injected using mutation operators (Abraham and Erwig 2009).

¹² No user interaction is needed in this simulation, because the expected, correct behavior is known by the simulation environment.

- (c) If the test case does not reveal a failure, we proceed with the next fragment.

At the end of the procedure, we know how many fragments have to be tested until the debugging method returns a set of potentially faulty formulas.

We tested two possible ways of ordering the fragments: (i) in random order, and (ii) in the order of their assumed complexity, and therefore, fault probability. Including this second ranking criterion helps us to assess the practical usefulness of the complexity heuristics described in Eq. 5. In the best case, our complexity heuristic would assign fragments the highest ranks that actually contain a fault. This, in turn, would mean that the user is able to locate an existing fault based on the outcomes of the algorithmic debugger after specifying the first test case.

5.2.2 Results

Table 4 shows the aggregated results of applying our experimental procedure to all examined spreadsheets. Since there are random factors in the fragmentation and test case generation process, the reported results are, again, the average of 100 simulation runs.

We report two main performance indicators for the two examined fragment ranking orders. The numbers are computed as follows:

1. *Number of formulas to be inspected* This number indicates how many formulas a user has to consider during test case specification until a set of potentially faulty formulas is identified by the debugger. Let us, for example, consider a simple spreadsheet that contains 10 formulas. The fragmentation process ends up with two fragments, where fragment $frag_1$ contains 3 formulas and $frag_2$ the remaining 7 formulas. Let us furthermore assume that $frag_1$ actually contains a fault. If the user starts testing $frag_1$ and the fragment test case reveals a failure, only 3 formulas have to be considered by the user when checking if the computed outcomes correspond to the expected ones. If the user starts with $frag_2$, test cases for both fragments are needed to detect the fault. The user therefore has to consider all 10 formulas at least once, which is the same as if the user had specified and checked a test case for the entire spreadsheet without fragmentation. For a fragment containing a fault, we assume that the user had to consider all formulas of the fragment to specify the expected values, since we cannot assume that inspecting only a single formula is enough to determine the expected value for its cell.
2. *Ranking quality* This number indicates how well our complexity heuristic is suited as a fault probability measure. Let us assume that our fragmentation method led to three fragments: $frag_1$, $frag_2$, and $frag_3$, with $frag_3$ actually containing a fault. Any optimal ranking would therefore have $frag_3$ on top of the list, i.e., $frag_3$ should be tested first. When determining the ranking quality, we assign a “ranking optimality” value of 100% in case the complexity heuristic ranked a fragment containing a fault on top of the list. Vice versa, a score of 0% was assigned, if the only faulty fragment was ranked at the bottom of the list. The percentage scores for other situations were assigned accordingly, depending on the position of the

Table 4 Overall results for the tested spreadsheets from the Enron Errors Corpus (Study II)

Ranked by	Avg. number of inspected formulas	Ranking quality (%)
Complexity	11.23	65.15
Random	14.60	51.75

first faulty fragment in the list, using the score $q_f = (|F| - r_f) / (|F| - 1)$ with r_f being the actual ranking of the fragment f .

The results in Table 4 show that independent of the chosen ranking strategy, only a small number of the formulas had to be inspected to find a fault.¹³ Even when using a random order to examine the fragments, on average less than 15 formulas of the 875 average formulas had to be considered during the test case specification process. Inspecting the fragments in the order of their assumed fault probability led to a further decrease of this number to about 11 formulas that have to be considered, which corresponds to 1.28% of all formulas of the tested spreadsheets. The *ranking quality* measure, likewise, confirms that the proposed complexity heuristic is a plausible proxy for fault probabilities, because the ranking quality was statistically significantly¹⁴ better when using the complexity heuristic than when randomly sorting the fragments. The number, however, indicates that there is room for further improvement. The detailed statistics for the individual spreadsheets are reported in Table 5.

5.2.3 Summary and limitations

The experiment results indicate that the fragment-based testing and debugging approach, when combined with complexity estimates, on average results in a faster localization of the faults in a given spreadsheet. Only in a few cases, a randomized testing and debugging process led to slightly better results.

While the capability of correctly ranking fault candidates is a common measure for comparing algorithmic fault localization procedures, a main criticism of such approaches is that those measurements are based on different assumptions. In particular, it is assumed that the developer goes through the list of candidates returned by the debugger from top to bottom; in addition, the measure implies that the developer immediately recognizes a fault when provided in a list of candidates. In reality, these assumptions may not always hold (Parnin and Orso 2011). Similar to Study I, a further limitation of this study is that the experiments were made only on a comparably small set of spreadsheets.

5.3 Study III: validation through a laboratory study

We finally conducted a laboratory study to assess the following research questions, based on the observed behavior of users when testing a spreadsheet.

¹³ Remember that without fragmentation users have to consider *all* formulas of a spreadsheet when manually calculating the outcomes for a given set of inputs.

¹⁴ We use the Mann-Whitney U test with $\rho = 0.05$.

Table 5 Detailed results for Study II

ID	Generated fragments	Position of faulty fragment		No. of inspected formulas	
		Complexity	Random	Complexity	Random
1	3.0	3.0	2.0	3.0	2.0
2	3.0	1.5	2.1	1.5	2.1
4	7.5	1.8	4.4	7.6	13.4
5	15.0	3.0	8.3	10.0	13.7
6	39.8	7.8	21.0	15.4	27.0
8	9.0	6.4	5.1	11.4	7.7
9	27.4	5.9	13.8	20.3	21.9
10	4.0	3.4	2.6	6.4	4.1
11	12.0	2.0	6.7	6.0	12.9
12	11.0	7.8	6.4	32.9	20.4
13	7.0	2.0	2.1	3.0	2.4
18	3.0	1.0	2.1	1.0	2.1
19	9.0	2.0	5.3	8.0	10.4
20	49.3	3.1	28.0	10.1	39.3
21	25.7	23.0	13.2	42.3	23.5
23	64.9	13.2	36.4	15.1	37.8
25	8.0	3.0	4.3	7.0	6.7
26	68.1	9.7	37.5	16.0	45.1
27	9.0	2.0	3.2	2.0	3.2
28	4.0	3.5	2.4	3.5	2.4
30	25.0	10.0	8.2	10.9	8.4

RQ1: Does the fragmentation approach increase the *efficiency* of testing, i.e., are users faster?

RQ2: Does the fragmentation approach has the potential to increase the *effectiveness* of testing, i.e., are the test cases better suited to find faults?

RQ3: How do users perceive and assess that general *utility* of fragment-based testing?

An additional goal of the study was to compare the usefulness of the results of the automated fragmentation approach to a situation where users define the fragmentation manually.

5.3.1 Experiment design

We used a between-subjects experiment design where the participants' task was to specify test cases for a given spreadsheet. The participants were randomly assigned to one of three groups.

- *AutoFrag* Participants of this group were instructed to initiate an automated fragmentation before starting to test the spreadsheet.

- *ManualFrag* Participants of this group were instructed to manually create a fragmentation before starting the testing process.
- *Control* Participants of the control group tested the spreadsheet without fragmentation support.

General procedure The participants were asked to accomplish the following tasks.

1. First, they had to fill out a pre-study questionnaire on their demographics, education and skills regarding spreadsheets and programming in general.
2. Next, they were handed out written tutorials, which were different for each participant group. The tutorials of the participants of the *AutoFrag* and *ManualFrag* groups contained information about how to create fragments with the tool manually and how to create test cases for the fragments. The tutorial of the control group only showed how test cases can be created with the tool. All tutorials contained hands-on tasks for the participants to get acquainted with the add-in.
3. The third and main task was specify test cases for the given spreadsheet. The intended logic of the spreadsheet was provided to the participants as a handout. The participants were instructed to use our add-in to create at least one test case for each fragment (groups *AutoFrag* and *ManualFrag*), or at least one test case for the entire spreadsheet (*Control* group). To specify the test cases, the participants had to manually provide input values (for a fragment or the entire spreadsheet) and label at least one output cell as being erroneous or correct. The participants could decide by themselves to end the task when they had the feeling that they had sufficiently tested the spreadsheet.
4. Once those participants who were provided with fragmentation support finished the main task, they were asked to repeat the previous step using the alternative fragmentation approach, i.e., participants of the *AutoFrag* group were asked to create the fragments manually and vice versa. Note that we excluded the observations that were made during this additional task when answering RQ1 and RQ2, since there might be learning effects.
5. All participants finally filled out a post-task questionnaire about their experiences during the process.

Test spreadsheet and seeded faults The spreadsheet used for the debugging task is shown in Fig. 6. We seeded *four* faults into the spreadsheet, explained in Table 6. The faults were designed in a way that the automated input coloring of our add-in would not reveal them immediately. The seeded faults belong to different general types of problems that were identified in the literature (Panko and Aurigemma 2010).

Participants and study environment We recruited 79 participants for the study which were mainly students of computer science or business administration at two universities in Austria. The participants had an average age of 24 years. Half of them reported to have at least some programming experience. The average self-reported experience with MS Excel across all participants was at about 4 on a scale from 1 (amateur) to 10 (expert). All participants completed the tasks in computer labs of the researchers' universities where the required software was pre-installed. All instructions for the participants were provided as hand-outs in order to avoid any bias introduced by the experimenter who was present in the room during the experiment. At the end, 57 participants successfully completed all tasks: 18 subjects of the *ManualFrag* group,

	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
2		Profit calculation															Free banana giveaways			
3																		Sold	196	
4																		Giveaways	20	
5																				
6	Product information			Sales numbers												Totals per product				
7	Product	Production cost	Retail price	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Sales	Revenue	Prod. cost		
8	Bananas	\$1	\$4	16	18	22	13	15	11	9	18	17	19	21	17	196	\$784	\$196		
9	Apples	\$2	\$4	12	22	23	18	16	10	18	22	23	18	16	10	208	\$832	\$416		
10	Cherries	\$2	\$3	8	12	15	20	16	14	39	34	22	18	26	42	266	\$798	\$532		
11	Pears	\$2	\$4	8	10	19	8	9	7	12	15	13	9	11	2	123	\$492	\$246		
12	Strawberries	\$1	\$3	1	0	0	10	15	19	33	12	0	0	0	0	89	\$267	\$89		
13	Grapes	\$1	\$2	21	19	12	10	8	10	16	11	18	12	8	2	147	\$294	\$147		
14	Oranges	\$3	\$4	1	5	0	0	0	0	0	0	10	18	22	26	82	\$328	\$246		
15	Blueberries	\$1	\$3	12	15	12	18	12	9	12	19	15	19	11	5	159	\$477	\$159		
16	Peaches	\$3	\$5	3	6	4	5	8	1	2	4	5	6	3	5	52	\$260	\$156		
17			Total sales	82	107	107	102	99	81	141	135	123	119	118	109					
18	Rules of tax calculation															Overall figures				
19	Amount for basic tax		\$150													Sales	1322			
20	Basic fees		\$5													Revenue	\$4,532			
21	Basic tax rate		15													Prod. costs	\$2,040			
22	Amount for add. tax		\$2,500													Profit before tax	\$2,492			
23	Add. tax rate		20													Taxes	\$378			
24																Profit	\$2,114			
25																				

Fig. 6 Spreadsheet to test that was used in the experiment

Table 6 Characteristics of seeded faults

No.	Type	Description
1	<i>Omission error</i>	The formula in U8 misses a certain cell in the calculation.
2	<i>Range error</i>	The formula in S12 misses a sales number in the range specification.
3	<i>Omission error</i>	The formula in T21 does not include the production costs from cell U13.
4	<i>Logical error</i>	The tax calculation in cell T24 is faulty.

15 subjects of the *AutoFrag* group, and 24 subjects of the *Control* group. Of the other participants, 12 did not follow the instructions, 6 did not fully understand the task, and 4 encountered a known bug in our software add-in.

5.3.2 Results

RQ1—Efficiency As a measure for the efficiency of the fragment-based testing and debugging approach, we measured how long it took for the participants to specify the required test cases.¹⁵ To determine the task completion times, we manually inspected the screen recordings that were captured during the experiment with the consent of the participants. The resulting average times—including the times for sub-tasks for the two treatment groups—are shown in Table 7.¹⁶ Detailed information about the data distribution for each participant group is provided in Fig. 7

When considering the *overall* time for task completion, the participants who used the automated fragmentation functionality (*AutoFrag*) were about 4 min faster than users who manually created fragments (*ManualFrag*), and 5 min faster than the *Control* group. To test the observed differences for statistical significance, we used a

¹⁵ We only took the time of step 3 of the above-described general procedure.

¹⁶ The times for the sub-tasks do not necessarily sum up to the overall time, as some participants performed intermediate steps between the tasks.

Table 7 Average times required to test the spreadsheet. Times are given in minutes and seconds

Group	Fragmentation	Test case creation	Overall
<i>AutoFrag</i>	00:36	21:20	24:13
<i>ManualFrag</i>	08:43	18:32	28:45
<i>Control</i>	–	27:38	29:40

Fig. 7 Distribution of recorded time values per task and participant group

one-tailed Mann-Whitney U test¹⁷ with $\alpha = 0.05$. According to this test and after applying the Bonferroni correction for multiple measurements, the recorded time differences are statistically significant with the following p values: *AutoFrag* to *ManualFrag* $p = 0.012$; and *AutoFrag* to *Control* $p = 0.007$. The time difference between the *ManualFrag* and *Control* groups is, however, not significant ($p = 0.436$). Overall, we can conclude for *RQ1* that using fragments can help to speed up the test process for spreadsheets.

RQ2—Effectiveness Since the participants were only required to identify failures (and not faults) in the given spreadsheets, we used an indirect approach to assess the potential effectiveness of our fragment-based method. The measurement was made as follows. When providing a test case, the participants could mark individual cells with our tool, indicating that the calculation outcome is either correct or erroneous. To assess the effectiveness of the testing and debugging process, we counted how often participants *correctly* marked a cell as containing a wrong calculation outcome, i.e., when the cell exhibited a failure for the given test case. When such a labeling by the user was correct, we know that there must be at least one fault along the calculation chain of this cell. The provided test case therefore has the potential that the algorithmic debugging technique will return this fault as a candidate. As an indicator for the potential effectiveness of the provided test cases with respect to the fault localization problem, we counted the number of faults that were revealed either directly or indirectly by correctly labeled failures for each participant. Furthermore, we counted how many cells were marked as being faulty overall. The results of this measurements are shown in Table 8 and Fig. 8 provides detailed information about the data distributions.

¹⁷ A Shapiro-Wilk test with $p = 0.05$ indicated that the measurements for the *AutoFrag* and *Control* groups are not normally distributed.

Table 8 Average number of correctly labeled failures and cells marked as exhibiting failures per participant group

Group	Avg. no. of correctly labeled failures	Avg. no. of labeled cells
<i>AutoFrag</i>	2.47	4.6
<i>ManualFrag</i>	2.11	2.6
<i>Control</i>	2.50	9.7

Fig. 8 Distribution of recorded number of correctly labeled failures and cells labeled as exhibiting failures per participant group

The participants, on average, correctly labeled failures resulting from between 2.11 and 2.5 of the four seeded faults. Looking at the results of this particular measurement, we see slightly better performance of the *AutoFrag* group compared to the *ManualFrag* group. To assess the significance of the observed differences, we again used a one-tailed Mann-Whitney U test with $\alpha = 0.05$ (measurements for the *ManualFrag* and *Control* groups were not normally distributed). The comparison revealed that the groups are not significantly different, with the following p -values: *AutoFrag* to *ManualFrag* $p = 0.113$; *AutoFrag* to *Control* $p = 0.444$; and *ManualFrag* to *Control* $p = 0.095$. We can therefore not conclude that the *AutoFrag* method led to an improvement over the other approaches in terms of our chosen effectiveness measure.

However, Table 8 reveals that the participants of the *Control* group labeled many more cells as failures than users who were supported by fragments. This shows that, while the fragment-based test cases are equally effective in terms of our measure, this same level of effectiveness required a much smaller set of user inputs in the form of labeled cells.

As a side observation, we noticed that three participants of the *Control* group did not exclusively rely on test cases for fault localization, but also inspected some cell formulas one by one. We did not exclude those subjects as they also specified test cases.¹⁸ Generally, the manual inspection of formulas is probably an effective

¹⁸ Excluding these subjects does not change the outcomes of our study.

Fig. 9 Questionnaire results regarding the value of using test cases

alternative debugging technique, which is orthogonal to approaches that are based on test cases. The scalability of such an approach is, however, limited when it comes to larger spreadsheets.

RQ3—User perception aspects This aspect was measured through the post-task questionnaire. Note that these measurements were taken after the participants (except those of the *Control* group) had tried out the alternative fragmentation approach. Since there might be certain effects that depend on the order in which they have experienced the different fragmentation approaches, the observations have to be interpreted with care.

Value of test cases A number of questions concerned the general concept of test cases. All participants of the study were asked to assess the utility of test cases, as well as the perceived complexity of specifying them. The outcomes for the different groups are summarized in Fig. 9. One main observation is that all participants found the usage of test cases, on average, helpful and intuitive (with no statistically significant differences between the groups). Overall, we consider the results to be an indicator that debugging based on test cases is generally an appropriate approach for end users. This result is in line with previous research about spreadsheet testing, which concluded that users test their spreadsheets at least to some extent (Hermans 2013).

Value of fragments The next block of questions focused on fragmentation and was only asked to participants in the *ManualFrag* and *AutoFrag* groups. The answers of the respondents are summarized in Fig. 10. The general opinion of the participants with respect to the usability and usefulness of fragments was very positive. The participants of the *AutoFrag* group found the fragment creation process slightly easier, but the differences between all groups for all questions were not statistically significant.

Comparing manual and automated fragmentations Since the participants of the groups *ManualFrag* and *AutoFrag* had made first experiences with the alternative way of creating fragments—see step 4 of the general experiment procedure above—we asked them additional questions about the two approaches. The results are shown in

Fig. 10 Questionnaire results regarding the value of using fragments

Fig. 11 Questionnaire results regarding automatically generated and manually created fragmentations

Fig. 11. Both groups preferred the *functionality* of generating the fragmentation automatically over the manual approach. Nevertheless, the option for manual fragmentation was appreciated as well, scoring an average rating above 3.5 in both participant groups.

As for the outcomes of the fragmentation processes, the participants were on average more *satisfied* when the process was automated. The observed differences were statistically significant. On an absolute scale, the members of the *ManualFrag* group rated the automated fragmentation results even higher than their own, manually created fragmentations.

When finally asked to state their preference for one or the other method, the large majority of the participants preferred the automated approach, as shown in Table 9. The participants of the *ManualFrag* group even had a stronger preference for the automated approach, which might however be an effect of the specific order in which they experienced the two alternatives.

Table 9 Questionnaire results when asking for preference regarding the manual or automated construction of fragmentations

Preference		
Treatment group	<i>Manual</i>	<i>Automatic</i>
<i>Manual</i>	3	15
<i>Automatic</i>	5	10

Generally, whether one uses automated or a manual fragmentation is not a mutually exclusive choice in our approach, as users of our tool can, for example, manually adapt an automatically generated fragmentation. Furthermore, a spreadsheet developer can decide to provide test cases both for individual fragments as well as for the entire spreadsheet. In some cases, it might be easier to provide a test case for the entire spreadsheet; this can in particular be the case when the overall outcome of a complex calculation is easy to judge as being correct as faulty, but the results of the intermediate calculations are more difficult to classify.

In that context, one should also keep in mind that different types of test cases also have different power in terms of revealing failures. Certain types of interacting faults might, for example, only become visible when test cases are used that cover longer calculation chains in a spreadsheet. As a result, it can be advisable to rely on a mix of fragment-based test cases and cases that span larger areas of the spreadsheet.

5.3.3 Summary and limitations

Overall, the user study indicates that a fragment-based debugging approach—when supported with appropriate tools—can help users to locate and fix faults in spreadsheets. Moreover, introducing spreadsheet users to such tools is likely to improve their overall proficiency in, as well as their awareness for, quality assurance practices like testing and debugging.

One limitation of the study is that we only used one specific spreadsheet for the laboratory study. However, while the spreadsheet was artificially created, we designed it to resemble real-world spreadsheets. Furthermore, the choice of the injected faults was based on observations from the literature. The complexity of the used spreadsheet can be considered to be medium. While much more complex spreadsheets can be found, e.g., in the Enron spreadsheet corpus, much simpler spreadsheets exist in this and similar corpora as well. In order to keep the complexity manageable for the study participants, we conducted a number of pre-tests which resulted in a spreadsheet that can be inspected by average users in about 30 min.

The number and characteristics of the participants is another potential limitation of our study. Regarding the background of the participants—students in computer science and business administration—we argue that they at least represent some of the common groups of spreadsheet users in the real world. Like the participants of our study, some real-world spreadsheet users in practice have programming skills. In addition, the general expertise of spreadsheet users, like for our study population, can be limited when it comes to the more advanced functionality of modern spreadsheet environments. Generally, one factor that could influence the generalizability of

the findings is that some participants, in particular those with a computer science background, are probably more acquainted to decomposition-based problem solving approaches than the average spreadsheet user.

Overall, however, we consider the threats to the validity of the specific insights gained from the laboratory study to be limited. Clearly, further experiments with additional spreadsheets and other types of participants are needed to assess to what extent the insights of this study can also be observed for different set-ups. In general, we assess the results obtained from the user study to be encouraging.

6 Related work

We structure our review of related works in three groups: spreadsheet testing, spreadsheet debugging, and visualization approaches.

Spreadsheet testing Regarding the general practice of users in the area of testing, (AlTarawneh and Thorne 2016) investigated the risk that comes from spreadsheets in the context of scientific research. They conducted an online questionnaire with 17 neuroscience researchers who use spreadsheets for collecting and manipulating research data. While more than 85% of the participants performed at least some form of manual testing (self-revision, peer review, recalculating), none of them used a testing tool.

In order to improve the tool support for spreadsheet testing, (Rothermel et al. 1998) proposed the ‘What You See Is What You Test’ (WYSIWYT) methodology. In their approach, users can mark cells either as correct or incorrect depending on the cell’s value. The system then gives immediate visual feedback about the amount of cells that have already been tested. In our work, we adopt similar ideas and also visually indicate when cell values have been marked as faulty or correct.

In Burnett et al. (2002), an extension to the WYSIWYT approach was proposed, aiming at improving the scalability of the method when a spreadsheet comprises larger homogeneous grids that contain identical formulas. Their definition of a grid is closely related to the concept of copy-equivalent cells in our work and the *logical areas* from (Mittermeir and Clermont 2002). Similarly, as in Burnett et al. (2002), it is sufficient also in our approach that the user specifies test cases only for a *representative* of an area. One main difference of our approach is that we introduce the concept of column-row-relatedness as a heuristic when looking for copy-equivalent areas. In addition, in our approach, the areas containing identical formals are identified automatically and also possibly merged with other fragments.

The AutoTest (Abraham and Erwig 2006) tool automatically generates a minimal amount of test cases that are necessary to achieve a full coverage of all feasible calculation paths of a spreadsheet. The tool achieves its goal by using backward propagation and constraint solving. While we could apply AutoTest for the test case generation of our approach too, the simpler approach that we used to generate test cases randomly was sufficient.

Rust et al. (2006) developed a tool for supporting the test-driven development of spreadsheets, where—in contrast to our approach—test cases are written before the corresponding spreadsheet formulas are created. The tool allows the user to define

input and output cells and to enter expected values for these cells. Each test case can then be run on the developed spreadsheet to see if the formulas fulfill all the defined test cases.

Expector Hermans (2013) is a tool that manages built-in test formulas. Test formulas are used to validate a spreadsheet's outcomes, e.g., a formula that checks whether a column with percentage values sums up to 100%. Expector automatically identifies test formulas in spreadsheets, gives the users feedback about the test coverage of their spreadsheet, and informs them for which cells they should write additional test formulas in order to increase the spreadsheet's test coverage. In contrast to Expector, in our approach we aimed to not change the spreadsheet itself in any way in order to test it.

Poon et al. (2014) addressed the problem that specifying test cases can be tedious and difficult for complex spreadsheets, and proposed to use metamorphic testing (Chen 2010) as a possible solution. A metamorphic test case requires a basic test case with input values and expected output values as well as the specification of a metamorphic relation which allows to automatically derive new test cases from the given basic test case. A metamorphic relation can be an equation or any other property describing (parts of) the system under test. The problem of specifying new test cases is then solved by using the information of the basic test case and the relation to add new test cases. Our fragmentation-based approach also aims at helping users to test complex spreadsheets. But instead of providing additional means to specify test cases, we tried to help users by allowing them to focus on smaller parts of the spreadsheet. Nevertheless, the relations of metamorphic testing in some sense correspond to the collapsed, semantically equivalent calculations in our approach, for which one test case can be reused for all semantically equivalent calculations.

Almeida et al. (2016) developed a framework for spreadsheet acceptance testing named SS-BDD (SpreadSheet Behavior Driven Development). This framework allows the user to write scenarios consisting of three parts: 'Given' (representing the precondition), 'When' (the actual event), and 'Then' (the post-condition). Scenarios are automatically translated into executable scripts and applied to the spreadsheet under test. Finally, a test report is generated. In contrast to our approach, the user has to decide on his own which parts of the spreadsheet should be tested together, and the tool does not support the user in determining the input and output cells. Furthermore, while the test report informs the user about the passed and failed test cases, it does not provide information about the test coverage.

Spreadsheet debugging Reichwein et al. (1999) proposed to use slicing to support end-users in the process of spreadsheet debugging. Later on, (Hofer et al. 2015) built upon the ideas of slicing for spreadsheets and adapted spectrum-based fault localization to the spreadsheet domain. Going one step further, (Abraham and Erwig 2005) developed GoalDebug, a tool that not only pinpoints the potentially faulty cell, but also proposes simple repair suggestions. However, GoalDebug heavily depends on predefined inference rules. In contrast to our approach, it might fail to identify faulty cells which require different changes than the ones given in the inference rules.

The work that is the most closely related to the model-based approach described in this paper is CONBUG (Abreu et al. 2015). This constraint-based approach translates a spreadsheet into a value-based model. More recently, Hofer et al. proposed to use

the more abstract and therefore faster solvable but also less accurate dependency-based (Hofer and Wotawa 2014) and comparison-based (Hofer et al. 2017) models. In principle, our approach can be extended to use these models. However, since the proposed fragment-based diagnosis approach is already faster than the cell-based one, there is limited need for us to sacrifice accuracy for additional speedups in the diagnosis process.

Spreadsheet visualization In Hermans et al. (2011), Hermans *et al.* presented data flow diagrams as a new visualization approach to support users in understanding a spreadsheet. These diagrams visualize the dependencies of the cells on three different levels of detail, the worksheet level, the cell level, and a data block level. At the data block level, each worksheet is divided into smaller parts, similar to our fragmentation approach. In contrast to our work, the data blocks in Hermans et al. (2011) are rectangular blocks, which does not have to be the case in our approach, which aims to minimize the complexity of the fragmentation. Generally, while there is some similarity at this level with our approach, the goals of Hermans *et al.*, who focus on spreadsheet comprehension, and our work on algorithmic debugging are very different.

7 Summary

We proposed a new algorithmic testing and debugging approach for spreadsheets. The approach is based on decomposing a spreadsheet into smaller fragments which can be tested and debugged separately. Three different types of studies showed that the proposed method can both make the algorithmic debugging process faster and at the same reduce the effort required by the user.

As part of our future works, we plan to investigate (a) alternative heuristics for the decomposition process and (b) other ways of assessing the fault probabilities of fragments in order to better guide the user in the debugging process. For this latter problem, we will in particular explore the use of spreadsheet smells for fault prediction.

Acknowledgements Open access funding provided by Austrian Science Fund (FWF).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

- Abraham, R., Erwig, M.: Goal-directed debugging of spreadsheets. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), Dallas, TX, USA, pp. 37–44 (2005)
- Abraham, R., Erwig, M.: 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, pp. 43–50 (2006)
- Abraham, R., Erwig, M.: Mutation operators for spreadsheets. *IEEE Trans. Softw. Eng.* **35**(1), 94–108 (2009)

- Abreu, R., Hofer, B., Perez, A., Wotawa, F.: Using constraints to diagnose faulty spreadsheets. *Softw. Qual. J.* **23**(2), 297–322 (2015)
- Almeida, L., Cirilo, E., Barbosa, E.A.: SS-BDD: automated acceptance testing for spreadsheets. In: Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing (SAST 2016), New York City, NY, USA, pp. 5:1–5:10 (2016)
- AlTawneh, G., Thorne, S.: A pilot study exploring spreadsheet risk in scientific research. In: Proceedings of the Annual Spreadsheet Risks Conference (EuSpRIG 2016), London, United Kingdom (2016)
- Burnett, M., Sheretov, A., Ren, B., Rothermel, G.: Testing homogeneous spreadsheet grids with the “what you see is what you test” methodology. *IEEE Trans. Softw. Eng.* **28**(6), 576–594 (2002)
- Chen, T.Y.: Metamorphic testing: a simple approach to alleviate the oracle problem. In: Proceedings of the 5th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2010), Nanjing, China, pp. 1–2 (2010)
- Cunha, J., Fernandes, J.P., Ribeiro, H., Saraiva, J.: 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, pp. 202–216 (2012)
- Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., Zanker, M.: Hierarchical diagnosis of large configurator knowledge bases. In: Proceedings of the Joint German Austrian Conference on AI (KI 2001), Vienna, Austria, pp. 185–197 (2001)
- Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M.: Consistency-based diagnosis of configuration knowledge bases. *Artif. Intell.* **152**(2), 213–234 (2004)
- Hermans, F.: Improving spreadsheet test practices. In: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2013), Ontario, Canada, pp. 56–69 (2013)
- Hermans, F., Pinzger, M., van Deursen, A.: Supporting. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), Waikiki, Honolulu, HI, USA, pp. 451–460 (2011)
- Hermans, F., Pinzger, M., van Deursen, A.: Detecting code smells in spreadsheet formulas. In: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012), Riva del Garda, Trento, Italy, pp. 409–418 (2012)
- Hodnigg, K., Mittermeir, R.T.: Metrics-based spreadsheet visualization—support for focused maintenance. In: Proceedings of the 9th Annual Spreadsheet Risks Conference (EuSpRIG 2008), London, United Kingdom, pp. 79–94 (2008)
- Hofer, B., Wotawa, F.: Why does my spreadsheet compute wrong values? In: Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering, (ISSRE 2014), Naples, Italy, pp. 112–121 (2014)
- Hofer, B., Ribeiro, A., Wotawa, F., Abreu, R., Getzner, E.: On the empirical evaluation of fault localization techniques for spreadsheets. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013), Rome, Italy, pp. 68–82 (2013)
- Hofer, B., Perez, A., Abreu, R., Wotawa, F.: On the empirical evaluation of similarity coefficients for spreadsheets fault localization. *Autom. Softw. Eng.* **22**(1), 47–74 (2015)
- Hofer, B., Nica, I., Wotawa, F.: AI for localizing faults in spreadsheets. In: Proceedings of the 29th International Conference on Testing Software and Systems (ICTSS 2017), St. Petersburg, Russia, pp. 71–87 (2017)
- Jannach, D., Engler, U.: Toward model-based debugging of spreadsheet programs. In: Proceedings of the 9th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2010), Kaunas, Lithuania, pp. 252–262 (2010)
- Jannach, D., Schmitz, T.: Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. *Autom. Softw. Eng.* **23**(1), 105–144 (2016)
- Jannach, D., Schmitz, T., Hofer, B., Wotawa, F.: Avoiding, finding and fixing spreadsheet errors—a survey of automated approaches for spreadsheet QA. *J. Syst. Softw.* **94**, 129–150 (2014)
- Mittermeir, R., Clermont, M.: Finding high-level structures in spreadsheet programs. In: Proceedings of the Working Conference on Reverse Engineering, pp. 221–231 (2002)
- Panko, R.R., Aurigemma, S.: Revising the Panko–Halverson taxonomy of spreadsheet errors. *Decis. Support Syst.* **49**(2), 235–244 (2010)
- Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA '11, pp. 199–209. (2011). <https://doi.org/10.1145/2001420.2001445>
- Poon, P., Kuo, F., Liu, H., Chen, T.Y.: How can non-technical end users effectively test their spreadsheets? *IT People* **27**(4), 440–462 (2014)

- Reichwein, J., Rothermel, G., Burnett, M.: Slicing spreadsheets: an integrated methodology for spreadsheet testing and debugging. In: Proceedings of the 2nd Conference on Domain-Specific Languages (DSL 1999), Austin, TX, USA, pp. 25–38 (1999)
- Reiter, R.: A theory of diagnosis from first principles. *Artif. Intell.* **32**(1), 57–95 (1987)
- Rothermel, G., Li, L., Dupuis, C., Burnett, M.: 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, pp. 198–207 (1998)
- Roy, S., Hermans, F., van Deursen, A.: Spreadsheet testing in practice. In: Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2017), Klagenfurt, Austria, pp. 338–348 (2017)
- Russell, S., Norvig, P.: *Artificial Intelligence: A Modern Approach*, pp. 126–129. Prentice Hall, Upper Saddle River (2009)
- Rust, A., Bishop, B., McDaid, K.: Test-driven development: can it work for spreadsheet engineering? In: Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2006), Oulu, Finland, pp. 209–210 (2006)
- Ruthruff, J.R., Prabhakararao, S., Reichwein, J., Cook, C., Creswick, E., Burnett, M.: Interactive, visual fault localization support for end-user programmers. *J. Vis. Lang. Comput.* **16**(1–2), 3–40 (2005)
- Scaffidi, C., Shaw, M., Myers, B.: Estimating the numbers of end users and end user programmers. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), Washington, DC, USA, pp. 207–214 (2005)
- Schmitz, T., Jannach, D.: Finding errors in the enron spreadsheet corpus. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2016), Cambridge, United Kingdom, pp. 157–161 (2016)
- Schmitz, T., Jannach, D.: An AI-based interactive tool for spreadsheet debugging. In: Showpieces presented at the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2017), Raleigh, NC, USA (2017)
- Schmitz, T., Hofer, B., Jannach, D., Wotawa, F.: Fragment-based diagnosis of spreadsheets. In: Proceedings of the 3rd International Workshop on Software Engineering Methods in Spreadsheets (SEMS 2016), Vienna, Austria, pp. 372–387 (2016)
- Schmitz, T., Jannach, D., Hofer, B., Koch, P., Schekotihin, K., Wotawa, F.: A decomposition-based approach to spreadsheet testing and debugging. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2017), Raleigh, NC, USA (2017)
- Shchekotykhin, K., Schmitz, T., Jannach, D.: Efficient sequential model-based fault-localization with partial diagnoses. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2016), New York City, NY, USA, pp. 1251–1257 (2016)
- Siddiqi, S., Huang, J.: Hierarchical diagnosis of multiple faults. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), San Francisco, CA, USA, pp. 581–586 (2007)

Affiliations

**Dietmar Jannach¹ · Thomas Schmitz² · Birgit Hofer³ ·
Konstantin Schekotihin¹ · Patrick Koch¹ · Franz Wotawa³**

Thomas Schmitz
thomas.schmitz@tu-dortmund.de

Birgit Hofer
bhofer@ist.tugraz.at

Konstantin Schekotihin
konstantin.schekotihin@aau.at

Patrick Koch
patrick.koch@aau.at

Franz Wotawa
fwotawa@ist.tugraz.at

¹ AAU Klagenfurt, Klagenfurt, Austria

² TU Dortmund, Dortmund, Germany

³ TU Graz, Graz, Austria