Improving Spreadsheet Test Practices

Felienne Hermans

Delft University of Technology

Abstract

Current testing practices for spreadsheets are ad hoc in nature: spreadsheet users put 'test formulas' in their spreadsheets to validate outcomes. In this paper we show that this practice is common, by analyzing a large set of spreadsheets from practice to investigate if spreadsheet users are currently testing. In a follow up analysis, we study the test practices found in this set to deeply understand the way in which spreadsheet users test, in lack of formal testing methods. Subsequently, we describe the Expector approach to extract formulas that are already present in a spreadsheet, presenting these formulas to the user and suggesting improvements, both on the level of individual test formulas as on the spreadsheet as a whole by increasing the coverage of the test formulas. Finally, we offer support to understand why a test formula is breaking. We end the paper with an example underlining the applicability of our approach.

1 Introduction

Spreadsheets are highly important to the business world; it is estimated that 95% of all U.S. companies use spreadsheets for financial analysis and that 90% of all analysts use spreadsheets for modeling. However, the use of spreadsheets is sometimes the cause of errors. In 2003, TransAlta lost US$24 Million because of an error in a spreadsheet[1]. More recently, the Federal Reserve made a copy-paste error in their consumer credit statement which, although they did not make an official statement about the impact, could have led to a difference of US$4 billion[2]. These stories, while anecdotal, underline the fact that more attention devoted to the correctness of spreadsheet could be beneficial for many companies.

In this paper we focus on supporting spreadsheet users in testing their spreadsheets, by leveraging and strengthening current testing practices in spreadsheets. Previous efforts in spreadsheet testing have mainly focused on either the automatic verification of spreadsheets or on supporting spreadsheet users in expressing tests or assertions, which could be subsequently verified by a spreadsheet systems, such as the WYSIWYT (What You See Is What You Test) spreadsheet testing system.

We start with the assumption that spreadsheet users already express tests, but that, since there are no testing platforms for spreadsheets, users use ordinary spreadsheet formulas to test with. These *test formulas* are formulas that differ from normal formulas, as they are used when a spreadsheet user wants to express a boundary or invariant, such as IF(A10<100,"OK","ERROR").

Firstly, we describe an approach to detect test formulas. With that approach, we val-

[1] http://bit.ly/cQRoy8

[2] http://bit.ly/6XwN9t

idate our assumption that test formulas occur frequent in the EUSES corpus [12], a well-known test set of spreadsheets. Since test formulas are indeed common, we perform a second, exploratory analysis on the EUSES corpus, in which we investigate their frequency, coverage and quality. From this analysis, we distill the most important problems with the current state of spreadsheet testing: 1) many formulas are not complete—i.e. they only consist of one branch, such as IF(A5=5,"OK")—, 2) coverage of the test formulas is often low and 3) support for understanding why a tests is breaking is often missing.

Subsequently, we describe an approach to solve these three problems. By analyzing the gathered test formulas and determining their completeness and coverage, our spreadsheet test tool Expector can support a spreadsheet user in improving their tests. Also, by generating names for the tests, Expector can give detailed feedback in the case a user broke a test.

We conclude with an example of our approach on one of the spreadsheets of the EUSES corpus, with which our approach is illustrated. With this example we show that focusing the attention of a spreadsheet user on test formulas can help to improve those tests and to catch errors early.

The remainder of this paper is structured as follows: In Section 2 we present an overview of related work on spreadsheet testing. Section 3 introduces the necessary background information, followed by Section 4 in which the selection of test formulas is treated. Section 5 and 6 describes the two analysis we performed on the EUSES corpus. The Expector approach is introduced in 7, while Section 8 explains its implementation. In Section 9 an example is described with which the applicability of our approach is illustrated. Limitations of this approach is discussed in Section 10, followed by the concluding remarks in Section 11.

2 Related Work

Improving the quality of spreadsheets is a subject of ongoing research, in which there are different directions to recognize. This section describes the most important of those directions.

Firstly, there is a set of papers that focuses on the unit correctness of spreadsheets, implemented in the UCheck tool [2, 8, 11]. Their units form a type system based on values in the spreadsheet, which is used to determine whether all cells in a column or row have the same type and is able to detect possible anomalies based on this type system. Ahmad *et al.* [3] also created a system to annotate spreadsheets, however their approach requires users to indicate the types of fields themselves.

Secondly, there are papers that use the code smells metaphor of Fowler [14] to locate suspicious formulas in a spreadsheet. In our previous work, we have investigated smells between worksheets [17] and within worksheets. Cunha *et al.* have also created a catalog of spreadsheet smells [9], which aims at smells in values, such as typographical errors and values that do not follow the normal distribution. Dig and Badame recently created a tool with which users can refactor spreadsheets [4].

Finally, there is the category of papers focused on testing spreadsheets. Burnett and Rothermel have written a series of papers on testing spreadsheets [5, 22]. With their *WYSIWYT*—Whay You See Is What You Test—method, spreadsheet users can explicitly mark cells of which they are sure their value is correct. With this information they calculate the 'testedness' of a spreadsheet and provide feedback to the user on how well a spreadsheet is tested. In empirical evaluations of the tool they showed that their system reaches an average fault detection percentage of 81% which is "comparable to those achieved by analogous techniques for testing imperative programs." [23]. In subsequent papers they have improved their method to work on large spreadsheets [13] and added support to help users test spreadsheets with multiple similar formulas [7]. A bit different in focus is the elegant work of Burnett *et al.* [6] that proposes to let end-users add assertions to their spreadsheets and generated 'system-generated' assertions by propagating the user-entered assertions through the dependency graph of the spreadsheet. In an experiment with these assertions, the authors found that assertions help end-users debug more effectively and more ef-

ficiently and that users were able to formulate assertions easily.

3 Background and motivating example

The related work on spreadsheet testing that we described in the previous section, however powerful, requires users to define their assertions or validate their formula outcomes in a somewhat new paradigm. We assert these methods have not yet caught wind in practice because of this necessary effort and paradigm shift. We have however seen that spreadsheet users do some testing, but do that in the form they are most familiar with: spreadsheet formulas.

Since most spreadsheet systems, including Excel, do not offer the possibility to test ones spreadsheets, spreadsheet users resolve to using ordinary formulas, when they feel the need to validate important outputs. We have seen in practice that this 'poor man's testing' has become a status quo. This method entails the use of conditional formulas like IF to express tests. Such a *test formula* is a formula of the form

IF(A1<16,"OK","margin cannot be over 16")

The intention of this formula clearly differs from a 'calculation' formula, like SUM(A1:A5). In the test formula, the user is using the conditional formula construction to express a test and some explanation on why the test could fail.

Figure 1 shows an example spreadsheet, from the EUSES corpus, where a test formula is used to validate that the sum of cells D6 to D10 sums to 100%. If this is the case, "100%" is outputted, if it is not, the formula shows "ERROR". These type of formulas are created by spreadsheet users to validate inputs and formulas. Sometimes, 'overall test formulas' are created that gather the results of all test formulas to indicate whether all tests have been passed. We state that the use of test formulas is common in practice and that test formulas differ enough from calculation formulas that they can be automatically detected.

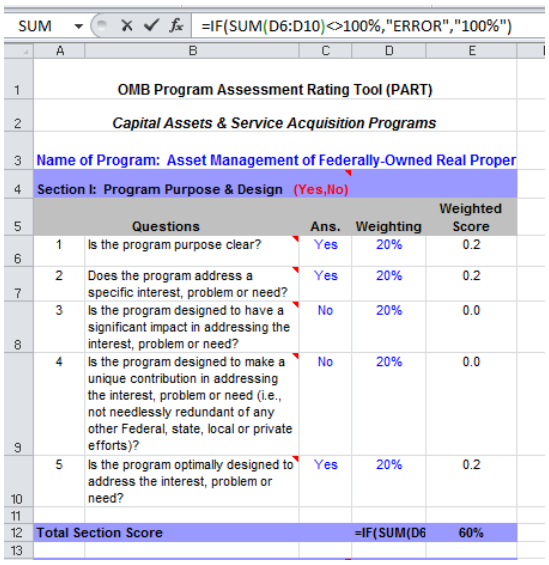

Figure 1: A spreadsheet with a test formula: The SUM of the values in D6:D10 should add up to 100%.

4 Selecting test formulas

In this section, we describe this approach to automatically distinguish between test formulas and calculation formulas.

The first characteristic of a test formula is the fact that the formula should have a conditional construction as root, to distinguish between a desired and an erroneous outcome. For this initial attempt, we only consider the simplest conditional construct: IF.

Secondly, at least one of the branches of this root IF should result in a string value (for instance "ERROR").

Finally, no other formulas should depend on a test formula. After all, test formulas should not be used for calculation. There is one exception to this criterion: a test formula may depend on other test formulas. This happens when an overall test is performed, as described in the previous section.

With this, we define the following characteristics of a test formula:

- A test formula has the operation IF as root function
- A test formula contains at least one branch

that can result in a string

- A test formula's result is not used in other formulas, unless these formulas are test formulas too

These conditions form the basis for our detection strategy.

5 Do spreadsheet users test? An evaluation on the EUSES corpus

In this section we describe an experiment in which we validate our hypothesis; in other words, we will analyze whether spreadsheets from practice indeed contain formulas that we classify as test formulas.

We test this hypothesis on the EUSES Spreadsheet Corpus [12]. This corpus consists of over 4000 spreadsheets collected from practice, from 11 different domains, ranging from educational to financial, and from inventory to biology. It was created in 2005 and has since then been used by several researchers to evaluate spreadsheet algorithms, for instance [1, 10, 15].

5.1 Experimental setup

To test our hypothesis, we use the Breviz spreadsheet analysis tool [16, 17, 18, 19] that we extended for this goal to detect test formulas in a spreadsheet. We ran Breviz on the 1,903 spreadsheets formulas in the corpus that contain formulas and outputted detected test formulas to a SQL Server database for analysis.

5.2 Results

Table 1 shows the results of this study; per domain, the second column lists the number of spreadsheets in which a test formula was detected, the third column shows the number of spreadsheets in the EUSES corpus that contain formulas.

Folder	with tests	with formulas	percentage
cs101	0	8	0%
database	10	216	4.6%
filby	4	35	11.4%
financial	24	383	6.3%
forms3	1	22	4.5%
grades	46	368	12.5%
homework	39	345	11.3%
inventory	23	303	7.6%
jackson	6	12	50%
modeling	13	207	6.3%
personal	2	4	50%
total	168	1903	8.8%

Table 1: Number of spreadsheets in the EUSES corpus containing test formulas

.

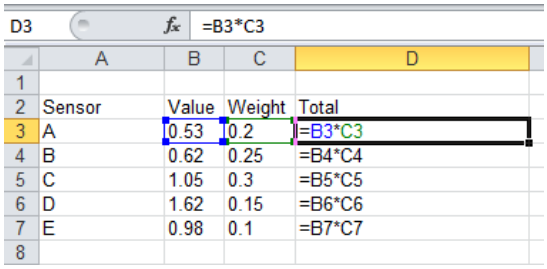

Figure 2: A spreadsheet containing 5 formulas, but only 1 unique formula.

In total, the EUSES corpus contains 12,295 formulas that we marked as test formulas out of a total of 90,468 formulas over the 1,903 spreadsheets with formulas. Counting in *unique* formulas, there are 567 unique test formulas over 5,785 formulas. Intuitively, a unique formula is the first formula of a column (or row) which has been copied through an entire column (or row). The spreadsheet in Figure 2 contains 5 formulas, however, this is only 1 unique formula. We determine the uniqueness of a formula by investigating its relative R1C1 notation. In the relative R1C1 notation, references to other cells are expressed relative to the cell containing the formula. In the example of Figure 2, all formulas have the relative R1C1 formula =RC[-2]*RC[-1] and hence, there is only one unique formula.

From these results we conclude that spreadsheet users do employ testing. This observa-

tion opens up possibilities to use the selected test formulas to improve spreadsheet development. Some of test techniques that exist for source code could now be applied to spreadsheets, such as test coverage calculation and regression testing.

6 How do spreadsheet users test? An evaluation on the EUSES corpus

Now we know that spreadsheet users test, we want to obtain a deeper understanding of their test practices, in order to be able to determine the right approach to support spreadsheet users in testing and even improve their practices. Therefore, we direct our attention back to the EUSES corpus and investigate the detected test formulas and research the current quality of the tests in the EUSES corpus.

6.1 Experimental setup

For this second, exploratory study, we specifically study the 168 spreadsheets that contained test formulas according to the study we described in Section 5. These spreadsheets together contain 109,997 formulas, of which 6,703 are unique.

6.2 Number of test formulas

The first measurement for test practice in the EUSES corpus is the number of tests per spreadsheet. Figures 3 shows the results of this first analysis. As can be seen in this figure, most spreadsheets (70 spreadsheets, which amounts to 42% of the spreadsheets) contain fewer than 10 test formulas. However, there is a long tail of spreadsheets with more tests, the most test formulas we encountered in one spreadsheet was 393. Figure 4 shows the number of unique test formulas per spreadsheet.

The total number of unique test formulas in the 168 spreadsheets with tests is 567. Figure 4 shows how these formulas are divided over the spreadsheets. As can be seen from this figure, the highest number of unique test formulas that any spreadsheet in the corpus has is 20.

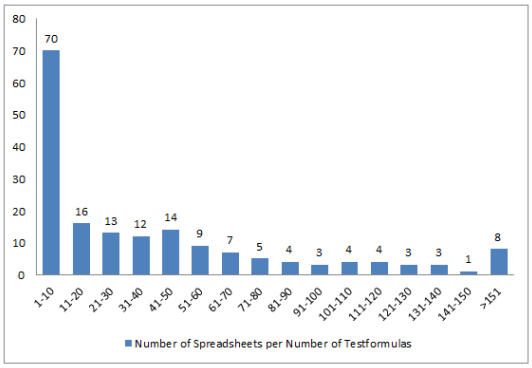

Figure 3: Distribution of test formulas over the 168 spreadsheets in the EUSES test set.

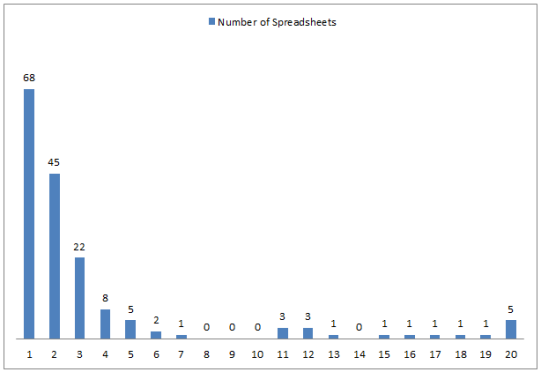

Figure 4: Distribution of *unique* test formulas over the 168 spreadsheets in the EUSES test set.

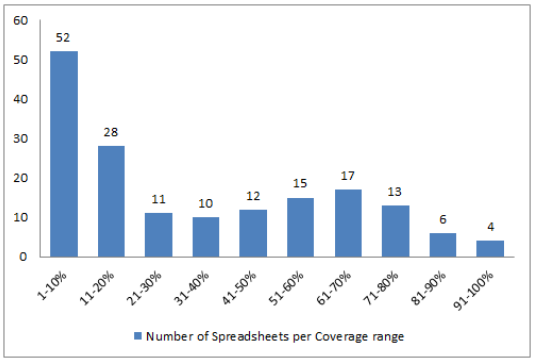

Figure 5: The number of spreadsheets for the different coverage percentage ranges.

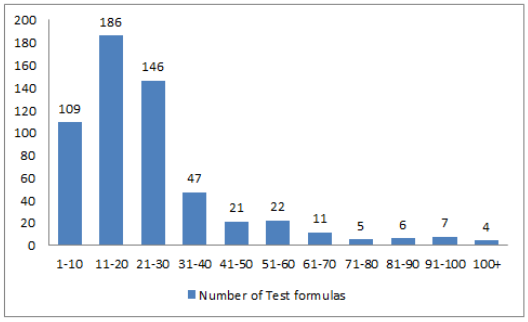

Figure 6: The unique formulas by average precedent distance.

6.3 Coverage in the EUSES corpus

Secondly, we investigate the formula cells that have been covered by a test formula. coverage. To calculate this *formula* coverage, we count the number of formulas in a spreadsheet that are direct or indirect precedents of a test, in other words, cells of which the results are used in a test formula, directly or indirectly. We call these formulas the 'tested' formulas' and obtain the formula coverage by dividing it by the total number of formulas in the spreadsheet. The formula coverage is a measure for how well the formulas in a spreadsheet are tested.

Figure 5 shows an overview of the results. In this figure, we see that while a large group of spreadsheets has a low coverage, there are also a number of spreadsheets that obtain very high coverages, up to 100%. That shows that some spreadsheet users are already thoroughly testing their spreadsheets, but there is quite some improvement in coverage to be made for spreadsheet users. While we did not interview the owners of the spreadsheets for this study, when analyzing the spreadsheets, it looks like spreadsheet users did not follow a structured method to test all formulas.

6.3.1 Difference with the WYSIWYT approach

In the WYSIWYT spreadsheet test paradigm[6], the 'testedness' of a spreadsheet is based on spreadsheet users mark formula outcomes as correct. Then, their system calculates what formulas led to this value and increases their testedness. A downside of this approach is the fact that the 'testedness' depends on the variation in the test cases of the spreadsheet, as there needs to be a cell that is pronounced correct by the user included in each path of calculation. Our coverage does not depend on input values, so it can even calculate the coverage of an 'empty' spreadsheet: a spreadsheet that contains formulas, but no inputs values for the formulas have been provided. We assert that the formulas form the heart of a spreadsheet model, and hence it is important that they are tested.

6.4 Location of tests

The next factor of testing we investigate is the location of tests. We want to know where spreadsheet users place their tests, to understand how they currently test their spreadsheet. For this, firstly, we look at the distance in the spreadsheet between the test and the tested cell as we want to know if spreadsheet users typically place their tests close to the cells they are testing. We calculate the distance between two cells as the sum of the difference between their rows and the difference between their columns. As such, the distance between A1 and C5 is 6, as C - A = 2 and 5 - 1 = 4.

To calculate the distance of a test formula, we have taken the average of the distance between the test formula and all its precedents.

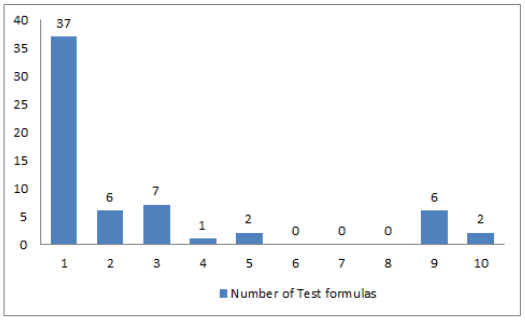

Figure 7: The number of worksheets outside of own worksheet that test formulas refer to.

Figure 6 shows the distribution of those averages. Note that the total number of test formulas in this graph sums up to 564 rather than 567, as there are three test formulas who only range over cells in a different worksheet, and for those, distance in undefined.

Interesting in this graph is the fact that the biggest category is not the lowest; in other words, the most test formulas are found quite close (average distance between 11 and 20) but not as close as possible. We do see that the vast majority of test formulas ($491 = 87\%$) has an average under 40. We suspect this has to do with the size of a spreadsheet on a screen. With standard screen size, resolution and font size, a spreadsheet user sees about 30 rows and 10 columns, so precedents with distance 40 would still be in sight, whereas higher distances would be outside of the view of the user. From these results, we conclude that it is important for spreadsheet users to see test and tested formulas together.

Regarding the location of test formulas, secondly we investigate the issue of test formulas ranging over multiple worksheets in more depth. As stated above, there were three formulas that directly referred to only cells in a different worksheet, like IF(Mileage!E6>0,“ ”,“Enter odometer reading on Mileage tab.”). However, there were 61 formulas (11% of all test formulas) that *indirectly* referred to cells in a different worksheet. Of these, Figure 7 shows an overview. In this figure, we can see that there are 37 tests referring to cells in 1 other worksheet, the other 24 test formulas refer to even more worksheets. We concur that these test specifically can be seen as *integration tests*, testing how worksheets in a spreadsheet work together.

6.5 Completeness of tests

Finally, we investigate the completeness of tests. Excel supports the use of ‘incomplete’ conditional formulas, these are conditionals in which only one of the branches has been filled. A formula like IF(A1<5,“ERROR”) is a valid Excel formula, which results in the boolean value FALSE if A1 is larger than or equal to 5. These incomplete formulas can be used as test formulas too. Furthermore, test formulas can be of the form If(A12<100,“OK”,“”) We consider this too to be an incomplete test formula, as the else branch does not provide us with information on why the test is failing. We assert it is better to test with compete conditionals, with a non-empty string expression for both the if and else branch, as that way it will be easier to understand why a test is breaking. For instance, if IF(A5<10,“OK”) fails, we will just get FALSE as result. Were we to add an else clause, like “10 is the maximum value” or even just “not OK” it is more clear that a test is breaking. We have surveyed the EUSES corpus and found that the use of incomplete test cases is quite common, in 93 out of the 567 unique formulas (16%), the else branch was either non provided or consisted of an empty string.

A second category of incomplete tests that we found were formulas that mixed a calculation with a test formula. An example of such a formula can be found in Figure 8. This formula is clearly a test formula, by outputting “??” the creator of this spreadsheet communicates to its user that a value below 65 is an unexpected result. However, for values 65 and higher, the result is the grade for the course. While this needs additional evidence, we think it is better to clearly separate test formulas from ‘normal’ formulas and suggesting to the user to put the test clause—in the example of Figure 8 this would be a formula IF(J2<65,“??”)—in a separate cell.

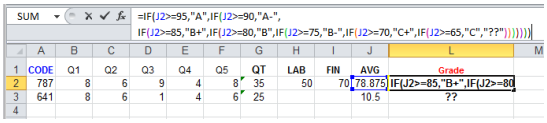

Figure 8: A formula with a test and a calculation component.

6.6 Summary

By studying the test practices in the EUSES corpus, we can identify a number of areas in which improvements could be made. Firstly, there is the improvement of tests that are already detected. Updating an incomplete formula, or changing a formula output from "not OK" to a more instructive message like "interest cannot be negative" is a small change of which a future user could benefit greatly. Secondly, there is the increase of coverage. This is a change that will require more effort from the user, as he has to think about the right tests, but a good system could support him, for instance by showing which formulas have already been tested and helping them to decide what additional formulas to test in order to increase coverage. Finally, with the more expressive tests, we can help users to understand why a recent change has broken a test.

7 The Expector approach

The studies described in the previous section show that, although spreadsheet users tests, testing practices could be greatly improved by supporting spreadsheet users to reach a more structured testing approach.

This section describes an approach to improve testing practices of spreadsheet users by analyzing the extracted test formulas. To that end, we follow the approach outlined in Figure 9.

7.1 Step 1: Selection of test formulas

The first step of the Expector approach is the selection of the test formulas. In this step all formulas are read by Expector and those formulas that comply with the test formula requirements as described in Section 4 are gathered.

7.2 Step 2: Determining completeness

In this step, Expector analyzes the completeness of the selected test formulas. Test formulas that have a branch that is either empty, like in the formula IF(B6=0,,"OK") or that have a branch which outputs an empty string or a string containing spaces are marked as incomplete.

7.3 Step 3: Distinguishing between failing and passing outcomes

In order to be able to reason with the extracted test cases, we need to be able to determine between a failing and a succeeding one. Since the formulas do not provide this information natively, we need to perform analysis on the outcomes. We do this by comparing the result value with a list of outputs associated with positive and negative results. We have created this list based on values encountered in the EUSES corpus. Words like "ERROR", "Not OK", "REJECT", "not correct" are linked to failing test cases, while results like "OK", "all ok", "validation passed" or ":):):)" signal passing test values.

Our Expector approach tries to map all branches of formulas marked as test formula to a result type (i.e. passing or failing). If only one of branch can be mapped, we assume the other branch has the opposite result type. This happens for instance when a branch results in a complex string, like IF(A1<>0,"OK","zero is never an allowed value for A1"). In this case we assume that the branch not resulting in "OK" represents a failing test case. Empty or missing branches in incomplete formulas are mapped to failing test cases.

After the initial suggestion has been done by Expector these suggestions can be viewed and altered by the user, via the interface shown in Figure 10.

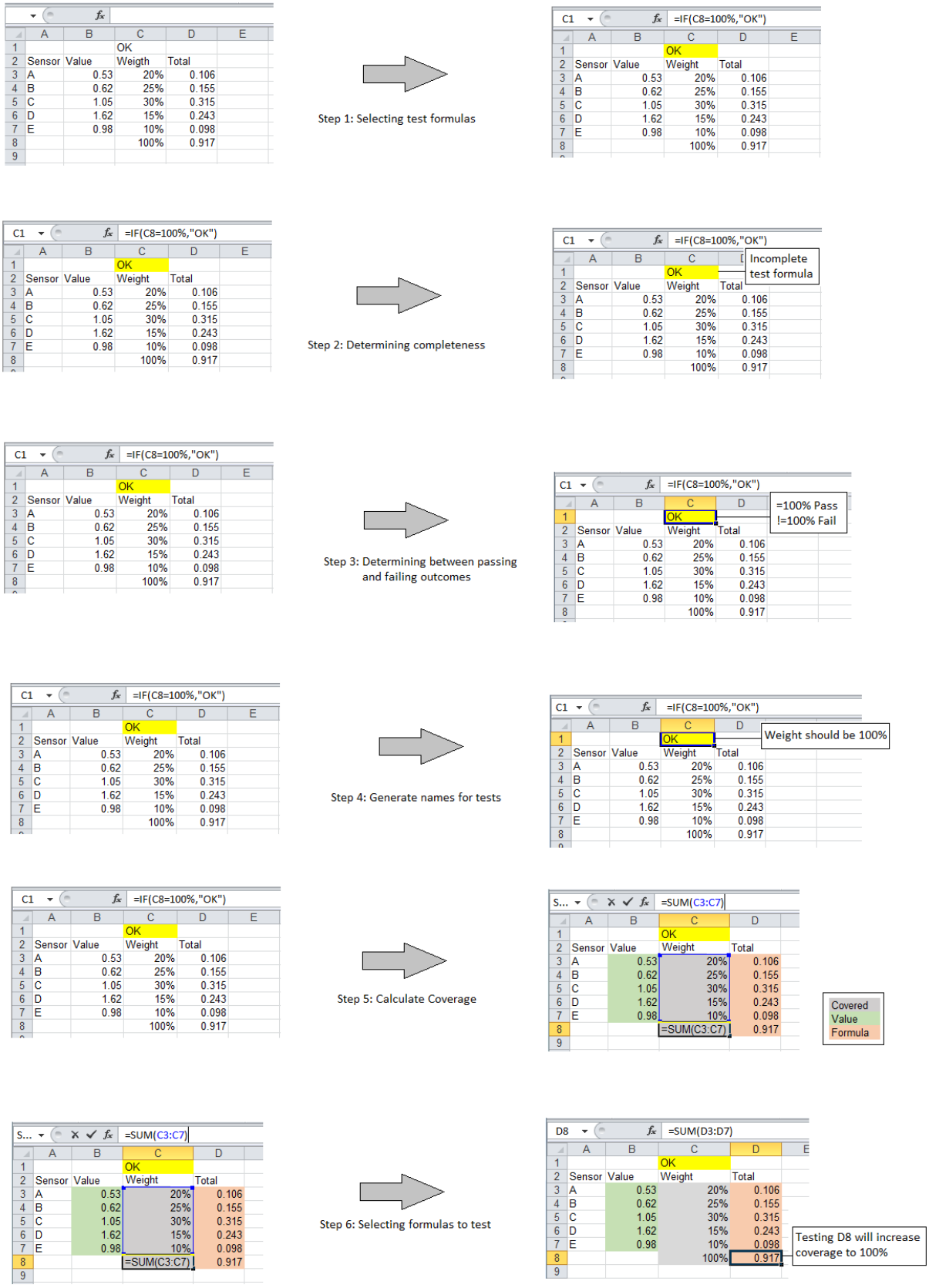

Figure 9: The six steps of the Expector approach: Selecting test formulas, determining completeness, determining failing and passing, name generation, coverage calculation and additional test selection.

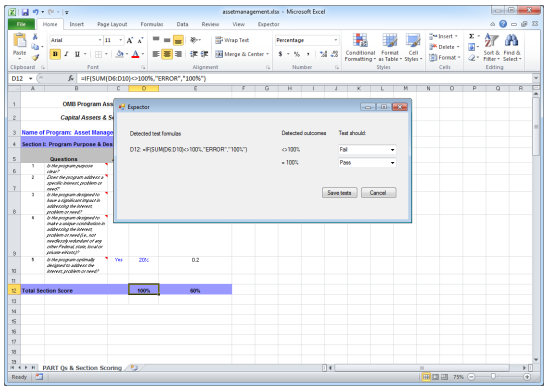

Figure 10: A screen shot of the Expector plugin. In this screen the user can verify the choice for failing and passing branches that has been made automatically.

7.4 Step 4: Generate names for tests

We want to help users to understand when a test is breaking and to be able to give a good explanation for this, we generate names for each test case. Names for test cases will be generated in the form of:

Referenced cell name "should be" passing rule for test

To generate descriptive names for the referenced cells, we use an approach based on our earlier work [15], in which we extracted class diagrams from spreadsheets. This method entails identifying row and column labels to generate labels for a cell. Here, we refine this approach slightly, by first analyzing whether a cell has an assigned named range. If so, the name of the named range is used. Otherwise, the approach of [15] is applied.

For instance, the name of the test in cell C1 in Figure 9 is 'Weight should be 100%', since 1) the name of C8 is 'Weight', as this is the column header and 2) the value 100% results in "OK", hence this is the clause that should hold.

7.5 Step 5: calculating the test coverage

In the fifth step of the Expector approach, we calculate the percentage of formulas that the extracted set of test formulas currently covers, with the method that was described in Section 6.3: we divide the number of tested formulas by the total number of formulas. In the example in Figure 9, this formula coverage is 50%, as there are 12 formulas: C3:C8 and D3:D8 of which 6 are covered by a test, C3 to C8.

7.6 Step 6: selecting the best formulas to test

In the final step of the Expector approach, the formulas are selected for which adding a test would increase coverage the most. To detect these formulas, we inspect all untested formulas and rank them by the number of untested formula precedents, i.e. cells containing a formula whose value is used in the untested formulas. The untested formula with the highest number of untested formula precedents will contribute the most to the formula coverage, as all its untested precedents will become tested if a test is added for this cell. In the example in Figure 9, the formula in D8 is selected as formula to test, as it refers to 5 untested formula cells: D3 to D7.

8 Implementation

Our Expector approach to extract and present test formulas Expector has been implemented as an Excel 2010 plugin and is written in C# 4.0. It uses the spreadsheet analysis core of our previous tool Breviz[16, 17]. It utilizes the Gembox component to read Excel files.[3]

Expector analyzes an Excel file and executes the above described test extraction algorithm and subsequently shows the user what tests have been detected, what the current formula coverage percentage is and what test formulas still need manual validation of passing or failing outcomes. Furthermore, Expector provides suggestions for the improvement of tests and the increase of coverage. Suggestions are shown in the the 'what's next' screen, in which the first improvement is listed that a user could perform to improve the spreadsheet's tests. Our approach favors making incomplete tests

[3] http://www.gemboxsoftware.com/spreadsheet/overview

complete over adding new tests, as this is the easiest improvement to make.

Finally, by generating names, Expector warns the user if a test is breaking. This is done by showing the name of the test and the conditional that has been broken in red below, see Figure 12. In addition to immediate feedback, it is also possible to view the outcome of all tests at once in the 'tests overview' screen. In future empirical tests, we plan to determine what type of feedback spreadsheets users prefer.

9 Example spreadsheet

This section describes how our Expector approach addresses the problems with spreadsheet testing that we found in the EUSES corpus: improving the quality of test formulas, increasing coverage and supporting test understanding. We illustrate our approach on one of the spreadsheets of the EUSES corpus.

9.1 Test run

We have studied the file Grades_suffix_19 from the folder Grades. A screen shot of this file can be found in Figure 11. The spreadsheet including our analysis can be downloaded from the authors webpage[4]. We have ran Expector on this spreadsheet and describe the suggested improvements in the following.

9.2 General outcome

Expector analyzed the given spreadsheet in 9.75 seconds on a Intel i5 1.70 Ghz processor with 4Gb of RAM. 46 formulas were detected of which 10 were marked as test formulas, these are the formulas in J2:J11. This leads to a coverage of 24%.

9.3 Improving test formulas

By clicking the 'what's next' screen, we find that Expector's first suggestion is to improve the selected test formulas, as they contain an empty branch. The first improvement we could make is to add an output value

[4]http://swerl.tudelft.nl/bin/view/FelienneHermans

J2 =IF(I2>90,"A",IF(I2>80,"B",IF(I2>70,"C",IF(I2>60,"D",IF(I2<60,"F",""))))

	A	B	C	D	E	F	G	H	I	J
1	Student	ID	PID	Test 1	Test 2	Test 3	Test 4	Total	PCT	Letter
2	Adams, J	312-34-5678	4-5678	80	86	82	85	253	84.33333	B
3	Boyd, S	435-67-8564	7-8564	90	88	75	0	253	84.33333	B
4	Carr, J	354-54-9867	4-9867	65	0	75	77	217	72.33333	C
5	Davis, M	987-56-7456	6-7456	77	82	0	80	239	79.66667	C
6	Evans, B	132-56-2435	6-2435	98	90	92	100	290	96.66667	A
7	Farris,W	897-76-8743	6-8743	67	76	75	76	219	73	C
8	Grant, S	345-24-8565	4-8565	86	80	0	79	245	81.66667	B
9	Hill, J	356-87-9845	7-9845	87	60	65	0	212	70.66667	C
10	Lake, P	132-45-3421	5-3421	90	92	90	98	280	93.33333	A
11	Stills, C	546-34-7643	4-7643	79	0	78	77	234	78	C
12	Perfect	111-11-1111		100	100	100	100	300	100	
13										
14	Number Taking Test			10	8	7	8			

Figure 11: The Grades_suffix_19.xls example that illustrates our approach.

and, for instance, change the formula into IF(I2>90,"A",IF(I2>80,"B",IF(I2>70,"C" ,IF(I2>60,"D",IF(I2<60,"F","ERROR"))))). But if we look more closely at the spreadsheet, we can see that a value above 100 in column I would still result in "A" and the user most likely would want an error value there, judging by the 'perfect' score in row 12. A test formula which catches values over 100 would improve the spreadsheet even more. Since our attention was drawn to this test formula, we start to investigate it more thoroughly. This illustrates a benefit of the Expector approach: it focuses the attention of the spreadsheet user on test formulas and the formulas that are being tested.

9.4 Selecting most contributing formulas to coverage

With respect to improving coverage, Expector can calculate where to add tests to increase the coverage the most. In the case study, Expector firstly suggests to add a test for I12, as this would cause I12 and H12 to become tested. While it might seem a bit too much to add a test for the 'perfect' row, doing so might improve the spreadsheet by adding additional information. We could, for instance, add the formula for calculating the mark and subsequently validate whether it results in "A" This will 1) help to prevent errors in the future, if the values in D12:G12 might be manipulated and 2)

express to the user of the spreadsheet that the grade "A" corresponds to the perfect score. In this way spreadsheet test, like unit tests or integration tests in software, can also serve as documentation.

When a test for I12 is added, Expector suggests to add tests for D14:G14. These formulas, that count the number of students that took a certain test, have not been tested yet. To test these, we could add a test comparing them to the total number of students or validate that all tests were taken by at least one student. After this test is added, only the formulas in C2:C11 remain untested. In these cells, the first 4 characters of the student ID are removed.

9.5 Warn users if a test is breaking

Finally, we investigate the usefulness of Expector in helping us to understand when we are breaking a test. Since we added a test expressing that the formulas in column I cannot exceed 100, when we enter a value causing this, for instance by entering 130 in D2 we get detailed feedback from Expector. As shown in Figure 12, in the bottom of the screen the reason of the breaking test is shown in red. In the interface, we choose a negotiation style—that allows the user to act when he wants—over an interruption style, where the user has to act immediately. In previous work, it has been shown that end-users are more productive in debugging with the negotiation feedback style [21].

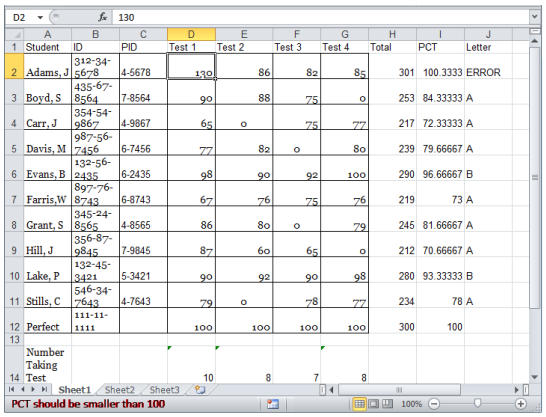

Figure 12: The Grades_suffix_19.xls where a test is breaking. This is indicated by the warning 'PCT should be smaller than 100' in the status bar.

10 Discussion and future work

The current Expector approach, while just a first prototype, is able to detect spreadsheets test formula and present them to users, and to support users in improving their tests. This section describes limitations to the current approach and directions for future research.

10.1 Extensive user study

In this paper, we have mainly focused on analyzing the problems around spreadsheet testing in the EUSES corpus and describing an approach to improve that practice. In future work, we will run a more extensive study with spreadsheet users to fully validate our approach.

10.2 Improving name generation

The approach for name generation that we currently use is simple: we just look at the cells that are referenced in the test formula. For instance, looking at the example in Figure 10, we just output the name of the cell that is referenced. The naming could be stronger, for instance if we also analyzing the calculations. In the example in Figure 11, we gather the name 'PCT' just by inspecting the label. By taking the formula, H2/H12*100, into account we could gather that this means percentage and is calculated with respect to the perfect score, which is calculated in H12.

10.3 Different types of test formulas

While examining the test formulas in EUSES corpus, we found that there are different types of test formulas. Some clearly express a standard boundary value above or below a cell results should never be, as the example in Figure 9, where a sum of percentages should sum to 100%. This is not a choice of the spreadsheet

user, but a 'fact of nature', so to say, in most cases, percentages should sum to 100.

In addition to such tests, we also found tests that validate a certain assumption or business rule, for instance, where an interest rate should never exceed 3%. In principle, it is possible that an interest rate does exceed 3%, but in this model the user wanted to express that this should never happen. These types of tests might benefit from different treatment in the future, since for such a test, a user likely wants to validate that it is consistent over a whole spreadsheet, or even over an entire collection of spreadsheets. Extracting and validating such 'business rule tests' is an interesting direction for future work.

10.4 Conditional formatting

In addition to using formulas for testing, Excel and other spreadsheet systems also provide the option to change the markup of a cell based on its value. This feature can also be used to test formulas in spreadsheets, for instance by coloring a cell red when an unexpected value is calculated. We did a small analysis in the EUSES corpus and found that conditional formating is not used that extensive, only 27 spreadsheets in the corpus contained formulas with conditional formatting. This is probably caused by the fact that not all spreadsheet users are aware of this feature. We could however add the option to add conditional formatting formulas to the test set automatically. However, determining what values are failing and passing might be a challenge. Red and green text or background values could be easily mapped to test failing and passing outcomes respectively. However, the results of conditional formatting can also be a whole spectrum from green to red making automated mapping more difficult.

10.5 Taking formula smells into account

Inspired by the methods of Hodnigg and Mittermeir [20], we could also suggest the user to test the most complex formulas in a spreadsheet first. This could, for instance be done by the methods we previously developed to find *smells* within [18] and between [17] worksheets. The smelly cells might be important to test, if not yet covered by a test formula, as we found in previous work that smelly formulas are perceived as more error-prone. In an empirical evaluation of formula smells, one of the subjects stated that "the chance of errors in such a long formula is so much bigger; when I find errors, it is almost always in long formulas" [18].

10.6 A dedicated test formula

In this paper we have investigate the use of testing by means of formulas that Excel offers, namely the IF formula. While useful for the purpose of testing, it is not perfect for it. A downside of this approach, for instance, is duplication in the output, as in the formula IF(A5=0,"rate is 0, OK","rate should not be 0"). It could be interesting in the future to offer spreadsheet users a dedicated test formula, where only a condition and a rationale are inputted and the negation is generated automatically. A formula like TEST(A5=0, "Rate check") might help users to see the difference between a test formula and a calculation formula. However, these dedicated formulas could also have negative effects, for instance because of the learning curve. Additional research is needed to investigate the challenges and benefits of such a special formula.

11 Concluding remarks

This paper aims at supporting spreadsheet users to test their spreadsheet. We firstly presented an analysis of the EUSES corpus to investigate if and how spreadsheet users are currently testing. As we found that spreadsheet users do test to some extent, have developed an approach to extract formulas that are already present in a spreadsheet, presenting these formulas to the user and suggesting improvements both on the level of individual test formulas as on the spreadsheet as a whole by increasing the coverage of the test formulas. Finally, we offer support to understand why a test formula is breaking.

11.1 Contributions

The contributions of this paper are the following:

- A method to select test formulas in spreadsheets (Section 4)
- An analysis of the occurrence of such spreadsheet test formulas in the EUSES corpus (Section 5)
- A second exploratory analysis of test practices in the EUSES corpus (Section 6)
- A description of a the Expector approach for selecting and utilizing test formulas in spreadsheets (Section 7)
- An implementation of this approach in a tool (Section 8) and an illustration of its use (Section 9)

References

[1] R. Abraham and M. Erwig. Inferring templates from spreadsheets. In *Proc. of ICSE '06*, pages 182–191, 2006.

[2] R. Abraham and M. Erwig. Ucheck: A spreadsheet type checker for end users. *Journal of Visual Languages and Computing*, 18:71–95, 2007.

[3] Yanif Ahmad, Tudor Antoniu, Sharon Goldwater, and Shriram Krishnamurthi. A type system for statically detecting spreadsheet errors. In *Proc. of ASE '03*, pages 174–183, 2003.

[4] Sandro Badame and Danny Dig. Refactoring meets spreadsheet formulas. In *Proc. of ICSM '12*, pages 399–409, 2012.

[5] M. Burnett, A. Sheretov, and G. Rothermel. Scaling up a what you see is wat you test methodology to spreadsheet grids. In *Proc. of VL '99*, pages 30–37, 1999.

[6] Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proc. of ICSE '03*, pages 93–103, 2003.

[7] Margaret M. Burnett, Andrei Sheretov, Bing Ren, and Gregg Rothermel. Testing homogeneous spreadsheet grids with the "what you see is what you test" methodology. *TSE*, 28(6):576–594, 2002.

[8] Chris Chambers and Martin Erwig. Automatic detection of dimension errors in spreadsheets. *Journal of Visual Languages and Computing*, 20:269–283, 2009.

[9] Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva Hugo Pacheco. Towards a catalog of spreadsheet smells. In *Proc. of ICCSA'12*, pages 202–216, 2012.

[10] Jácome Cunha, João Saraiva, and Joost Visser. Model-based programming environments for spreadsheets. In *Proc. of SBLP '12*, pages 117–133, 2012.

[11] M. Erwig. Software engineering for spreadsheets. *IEEE Software*, 26:25–30, September 2009.

[12] M. Fisher and G. Rothermel. The EUSES spreadsheet corpus: A shared Resource for supporting experimentation with spreadsheet dependability mechanisms. In *Proc. of WEUSE '05*, pages 47–51, 2005.

[13] Marc Fisher II, Gregg Rothermel, Tyler Creelan, and Margaret Burnett. Scaling a dataflow testing methodology to the multiparadigm world of commercial spreadsheets. In *Proc. of ISSRE '06*, pages 13–22, Raleigh, NC, USA, November 2006.

[14] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[15] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Automatically extracting class diagrams from spreadsheets. In *Proc. of ECOOP '10*, pages 52–75, 2010.

[16] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *Proc. of ICSE '11*, pages 451–460, 2011.

[17] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proc. of ICSE '12*, pages 441–451, 2012.

[18] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting code smells in spreadsheet formulas. In *Proc. of ICSM '12*, pages 409–418, 2012.

[19] Felienne Hermans, Ben Sedee, Martin Pinzger, and Arie van Deursen. Data clone detection and visualization in spreadsheets. In *Proc. of ICSE '13*, pages 292–301, 2013.

[20] Karin Hodnigg and Rolandt Mittermeir. Metrics-based spreadsheet visualization: Support for focused maintenance. *Proc. of EuSpRIG '09*, 2009.

[21] T. J. Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R. Ruthruff, Laura Beckwith, and Amit Phalgune. Impact of interruption style on end-user debugging. In *Proc. of CHI '04*, pages 287–294, 2004.

[22] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. What You See is What You Test: A methodology for testing form-based visual programs. In *Proc. of ICSE '98*, pages 198–207, 1998.

[23] Karen J. Rothermel, Curtis R. Cook, Margaret M. Burnett, Justin Schonfeld, T. R. G. Green, and Gregg Rothermel. Wysiwyt testing in the spreadsheet paradigm: an empirical evaluation. In *Proc. of ICSE '00*, pages 230–239, 2000.