Integrating Anomaly Diagnosis Techniques into Spreadsheet Environments

Daniel Kulesz
Institute of Software Technology
University of Stuttgart
daniel.kulesz@informatik.uni-stuttgart.de

Jonas Scheurich
University of Stuttgart
jonas.scheurich@gmx.net

Fabian Beck
Visualisation Research Centre
University of Stuttgart
fabian.beck@visus.uni-stuttgart.de

***Abstract*—Although spreadsheets are often faulty, end-users like them for their flexibility. Most existing approaches to spreadsheet diagnosis are fully automated and use static analysis techniques to find anomalies in formulas or methods to derive test cases without user interaction. The few more interactive approaches are based on values already present in spreadsheets as well. In our work, we advance the idea of testing spreadsheets with user-defined test scenarios but encourage visually aided creation of independent test cases by separating the definition of test scenarios from the specific values present in the spreadsheet—just like test code is separated from production code in professional software. We combine the testing approach with static analysis and integrate it into a common visual spreadsheet environment named SIFEI. It supports users in interactively creating, executing, and analyzing their own test scenarios with a number of visual markers. Findings from two qualitative studies indicate that the concept is suitable for casual spreadsheet users.**

I. Introduction

While end users love spreadsheets for their flexibility, several studies indicate that faults in spreadsheets are common [1]. Many end users are overconfident about the correctness of their spreadsheets and do not examine the output values calculated in spreadsheets sufficiently, even when using them for mission-critical decisions [1], [2]. The European Spreadsheet Risks Interest Group has collected dozens of cases[1] where this chain of circumstances has led to severe financial and reputational losses.

Although there are several promising suggestions for preventing spreadsheet faults in the first place (i.e., by reducing overconfidence), diagnostic approaches for identifying faults in existing spreadsheets are needed to check existing spreadsheets and because not all faults can be prevented. Most diagnostic approaches can be divided into three categories (they will be discussed in more detail in Section VI):

- Fully automated (e.g., [3], [4]): These tool-based approaches require only low user interaction. Typically, they rely on generic specifications or derive specifications themselves without asking users. They are able to indicate only “likely” faults or “smells”. Approaches that automatically generate test cases produce lower numbers of false positives than approaches that check spreadsheets using fault patterns.
- Partially automated (e.g., [5]): These tool-based approaches require considerable amounts of interaction because they rely on user-defined specifications. But typically, they have a higher chance of detecting semantic errors than fully automated approaches.
- Manual (e.g., [6]): These approaches can be executed even without tools and are comparable to design and code inspections in professional software development [7]. They are executed manually by experts. Formal inspection process definitions accompanied by checklists are typical representatives of such approaches. The efficiency of manual approaches can be boosted by tools that aid in (structure) comprehension or identify ‘high-risk areas’ to narrow the inspection scope.

In general, automated approaches tend to be the cheapest but least effective ones. Manual approaches promise the best results but are time-consuming and rely on experts who are hard to find even in larger organizations. Partially automated approaches provide a good compromise between the two extremes and offer the additional benefit of generating a valuable by-product: specifications—an artifact hard to encounter in the spreadsheet world.

Since all of these diagnostic approaches have their pros and cons, it appears worthwhile to combine the approaches in a beneficial way [8, pp. 48-49]. Considering that some of the underlying techniques require software engineering knowledge, the EUSES Consortium[2] proposes to encapsulate this knowledge into tools. The key challenge here is the integration of several diagnostic approaches into a single spreadsheet environment. We want to address this challenge by introducing a tool called SIFEI which extends the popular spreadsheet execution environment Microsoft Excel. Our main contributions are the following:

- We integrate a variety of diagnosis techniques into a spreadsheet environment and visualize combined findings without interfering with the original layout of the spreadsheet.
- We introduce an approach for the interactive creation of test scenarios within a spreadsheet. In contrast to previous approaches, it allows the unbiased specification

[1]http://www.eusprig.org/horror-stories.htm

[2]http://www.eusesconsortium.org

of scenarios to be done independently of values present in the spreadsheet.

In a user study, we investigated the usability of the approach and particularly studied whether users not familiar with programming understand our concept of spreadsheet testing. The results indicate that participants were successful in applying our approach, but they also show that there is room for improvement in terms of motivating users to perceive the benefits of the approach more quickly.

In the following, we first briefly discuss terminology (Section II) and the technical back-end of our approach (Section III). Then, we describe our contributions in detail (Section IV), present results from a user study (Section V), and finally discuss contributions and results in the light of related work (Section VI).

II. Terminology

Often, spreadsheet-related terms are used ambiguously by people in research and practice. Therefore, we want to clarify what we mean when we use such common terms.

A spreadsheet is a program that is contained in a spreadsheet file document (e.g., in Office Open XML which uses the “xlsx” ending for spreadsheets) and executed in a spreadsheet execution environment (e.g., Microsoft Excel). A spreadsheet is composed of cells which can contain values and formulas. One spreadsheet may be composed of several worksheets. A spreadsheet is executed each time formula recalculation is issued.

From a domain perspective, the spreadsheet cells can be divided into input cells (data and so-called “decision variables”), intermediate calculation cells and output cells. In most cases, output cells are numeric. However, there are exceptions: For instance, in spreadsheets that heavily use conditional formating, users might just interpret the colors of certain cells as relevant output.

When working with spreadsheets, end users can commit errors by accidentally taking an unwanted action like mistyping a number or linking a wrong cell when constructing a formula. Unless detected and corrected, committed errors manifest themselves as faults in the spreadsheet. In cases of fraud, we do not regard the user actions as “errors”, since they were taken on purpose, but the result is the same: the spreadsheet is faulty. The spreadsheet can also get faulty in cases of misconception where users do not understand the domain and, therefore, execute the wrong actions.

Apart from faults, spreadsheets may also contain qualitative defects such as high formula complexity or other types of “bad formula style”. Such defects can promote faults later on (e.g., during maintenance), but we do not regard them as faults unless they directly result in wrong output values. Output values are results that are interpreted by end users. Thus, even correct output values can be misinterpreted.

The effects of faults can sometimes be hidden and lead to visible malfunction regarding the output values only under certain conditions (e.g., only when the input values are within a certain range). In such cases of visible malfunction, we say that we observe a failure of a spreadsheet. Failures may—but do not necessarily have to—cause a real-world impact (e.g., wrong decision made).

Therefore, to detect errors, an analyst must observe a spreadsheet user working with the spreadsheet (live or by means of a recording). By issuing manual or automated inspections that do not execute the examined spreadsheet, an analyst can only find signs for faults or defects. Only when the spreadsheet is executed during the inspection, we call this “testing”: it can result in observable failures, but it cannot directly point to faults. We refer to the process of trying to locate a fault after observing a failure as “debugging”.

Just like doctors in the field of medicine, we use the notion of “diagnosis” to describe any activity that involves actions connected with the goal to detect any anomalies such as errors, defects, faults, failures, or problems. A diagnosis can be issued without the presence of any prior symptoms and is not guaranteed to find any anomalies.

III. Spreadsheet Inspection Framework

The front-end for spreadsheet diagnosis that we present in this paper is based on a technical framework called Spreadsheet Inspection Framework (SIF)[3]. SIF provides an implementation of various spreadsheet diagnosis techniques but does not contain a front-end for user interaction. SIF is written in Java and uses the Apache POI library to examine the spreadsheets. SIF supports, among others, the following types of diagnosis techniques which we will use later for the integration into a spreadsheet environment:

- a fully-automated static analysis technique and
- a partially-automated testing approach.

Currently, three fully automated static analysis techniques that can detect qualitative defects are implemented in SIF. These techniques indicate (i) formulas that have a high complexity, (ii) formulas that violate the reading direction (i.e. they refer to cells that are not left or above them), and (iii) formulas that contain hard-coded constants. The testing approach is based on user-defined test scenarios that are executed on the spreadsheet, comparable to regression tests in software engineering: given a set of input values, output values are computed and checked whether they conform to the output values or ranges specified in the test scenarios.

IV. Spreadsheet Environment Integration

The main contribution of this paper is the consistent visual integration of the heterogeneous diagnosis techniques in a spreadsheet environment. In particular, we implemented this integration for Microsoft Excel in a tool called Spreadsheet Inspection Framework Excel Integration (SIFEI)[4]. From a technical perspective, it is a layer that integrates SIF with Microsoft Excel and provides a front-end. SIFEI is written in C# and uses the Windows Presentation Foundation. SIFEI and SIF exchange XML-formated data over a TCP socket

[3]https://github.com/kuleszdl/Spreadsheet-Inspection-Framework
[4]https://github.com/kuleszdl/SIFEI

connection. This makes the overall system architecture extensible regarding the support of other document formats or the integration into other spreadsheet execution environments like LibreOffice.

This section presents the visual user interface of SIFEI. We first demonstrate how findings of the static analysis are visualized in the spreadsheet in a lean fashion without destroying the original layout and coloring of the spreadsheet. Then, the integration of the testing approach in the form of user-defined testing scenarios is discussed.

A. Visual Indication of Findings

We illustrate the capabilities of SIFEI for visualizing findings detected by the static analysis techniques SIF using a very simple example spreadsheet, a calculator for the volume of a smartphone. SIF reports two qualitative defects for this spreadsheet.

SIFEI adds a ribbon bar (Figure 1, A) and three components for communicating findings to the user in a comprehensible way: An overview with a list of findings in our side pane (Figure 1, B), markers highlighting cells with findings (Figure 1, C) and tooltip dialogs with further explanations of findings (Figure 1, D). Since SIFEI just adds a layer of icons to the spreadsheet, the user is able to return to the original view of the spreadsheet by turning off the additional layer at any time.

1) List of Findings: The list of findings is aggregated by type of finding and sorted by severity. Different groups of findings from static analysis techniques are discerned as well as findings from the testing approach (Section III). The first are divided by the cells they are referring to, the latter are divided by scenario; each scenario might refer to multiple cells that deviate from the specified behavior. Each group in the list consists of a headline, a description of the group, and a list of aggregated findings that can be expanded on demand. Each individual finding in turn is described in further detail and carries a severity value that is user-defined for each group. The severity of the group of findings as displayed in the upper right corner is the sum of the severity of the individual findings. A bar at the side of each group in the list visually encodes the summed severity using a color scale from yellow (low severity) to red (high severity). Groups of findings as well as individual findings can be activated and deactivated. Also, findings can be marked as false positives and be hidden; this is important for findings gathered using static analysis techniques as these often overestimate defects but can be useful for temporarily disabling test scenarios that need rework as well. When the user clicks on an individual finding, the spreadsheet view jumps to the respective cell.

2) Cell Markers: Each cell in the spreadsheet that is affected by a finding is highlighted. Since coloring is often already applied by the users to mark other things in the spreadsheet, we decided to use small warning icons for this purpose instead; these are also easy to detect by the user but do not conflict with the original coloring of the spreadsheet. Each icon is colored according to the severity of the group of findings it belongs to. This color-coding already guides the user to the more severe findings and visually connects the findings in the list with those indicated in the spreadsheet.

3) Tooltip Dialogs: Hovering over a warning icon with the mouse pointer shows a description of the finding in a tooltip dialog. The dialog explains why the finding was raised and how the actual behavior or value for the highlighted cell deviates from the expected behavior. An alternative for retrieving details is clicking on the icon—then, the finding is highlighted in the side bar with a list of findings.

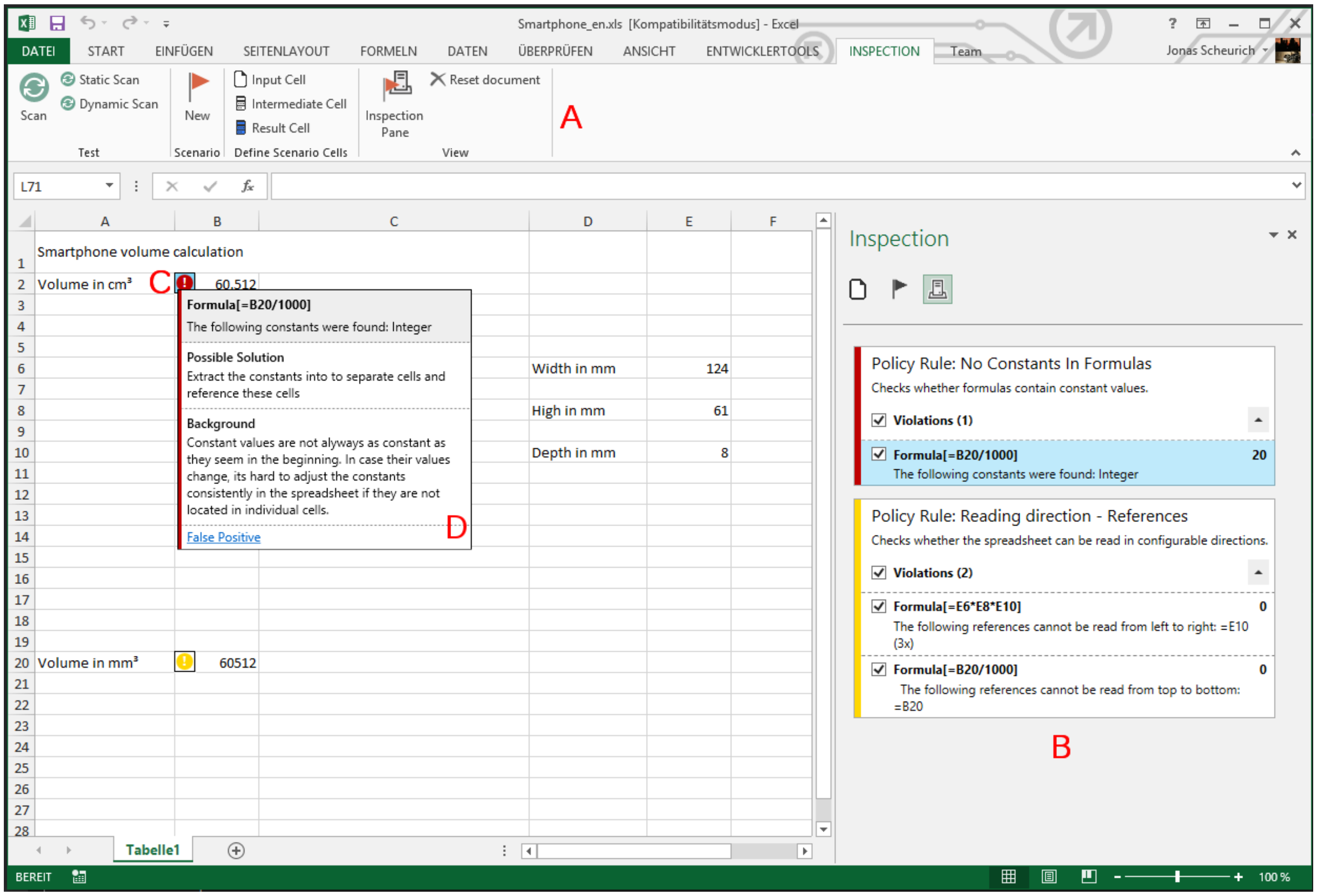
B. Testing Integration

In addition to providing a visualization for findings in spreadsheets, we also designed an interactive testing approach that allows for the specification of test scenarios. Next, we motivate the design decisions behind our approach and describe the conducted user interface for editing test scenarios in detail.

1) Testing Approach: As part of SIF, we developed an approach that merges the black-box concept of system testing with the automatic execution of unit tests. It views a spreadsheet from a purely functional perspective based entirely on test scenarios specified by the user. This is done using the following interactive workflow:

1) The user marks cells as input, intermediate, and output cells. Input cells are cells that the user expects to change himself depending on the current use of the spreadsheet while output cells are cells that the user expects to contain results from calculations. More complex calculations are usually broken down using intermediate cells and allow the user to understand how the calculation arrived at the output. This cell classification is the user's pure domain view and completely decoupled from the calculation chain in the spreadsheet.
2) The user specifies test scenarios by filling the previously marked input cells with values and providing expected values for intermediate and output cells. While input values must be specified precisely, expected values for output and intermediate cells can be defined as ranges as well. However, the user can freely choose how complete the specification of a test scenario shall be. Even providing just one input and one expected value is sufficient.
3) For each test scenario, a behind-the-scenes copy of the spreadsheet is opened, populated with the input values, and evaluated. The actual output values are compared with the expected output values from the test scenarios.
4) The described visual presentation reports any deviation in comparison between actual and expected output values to the user.

This workflow can be executed once or iterated multiple times refining and extending test scenarios while fixing reported deviations. However, just like in software engineering for traditional software, a detected deviation does not necessarily indicate a fault but can be a false positive due to the misspecification of the test scenario. This case would require fixing the specification of the test scenario. Nevertheless, spreadsheets would not be very flexible if they were not



	A	B	C	D	E
1	Smartphone volume calculation				
2	Volume in cm³	60.512			
6				Width in mm	124
8				High in mm	61
10				Depth in mm	8
20	Volume in mm³	60512			

Formula[=B20/1000]
The following constants were found: Integer

Possible Solution
Extract the constants into to separate cells and reference these cells

Background
Constant values are not alyways as constant as they seem in the beginning. In case their values change, its hard to adjust the constants consistently in the spreadsheet if they are not located in individual cells.

False Positive

Inspection

Policy Rule: No Constants In Formulas
Checks whether formulas contain constant values.
Violations (1)
Formula[=B20/1000] 20
The following constants were found: Integer

Policy Rule: Reading direction - References
Checks whether the spreadsheet can be read in configurable directions.
Violations (2)
Formula[=E6*E8*E10] 0
The following references cannot be read from left to right: =E10 (3x)
Formula[=B20/1000] 0
The following references cannot be read from top to bottom: =B20

Fig. 1. SIFEI's visualization of findings.

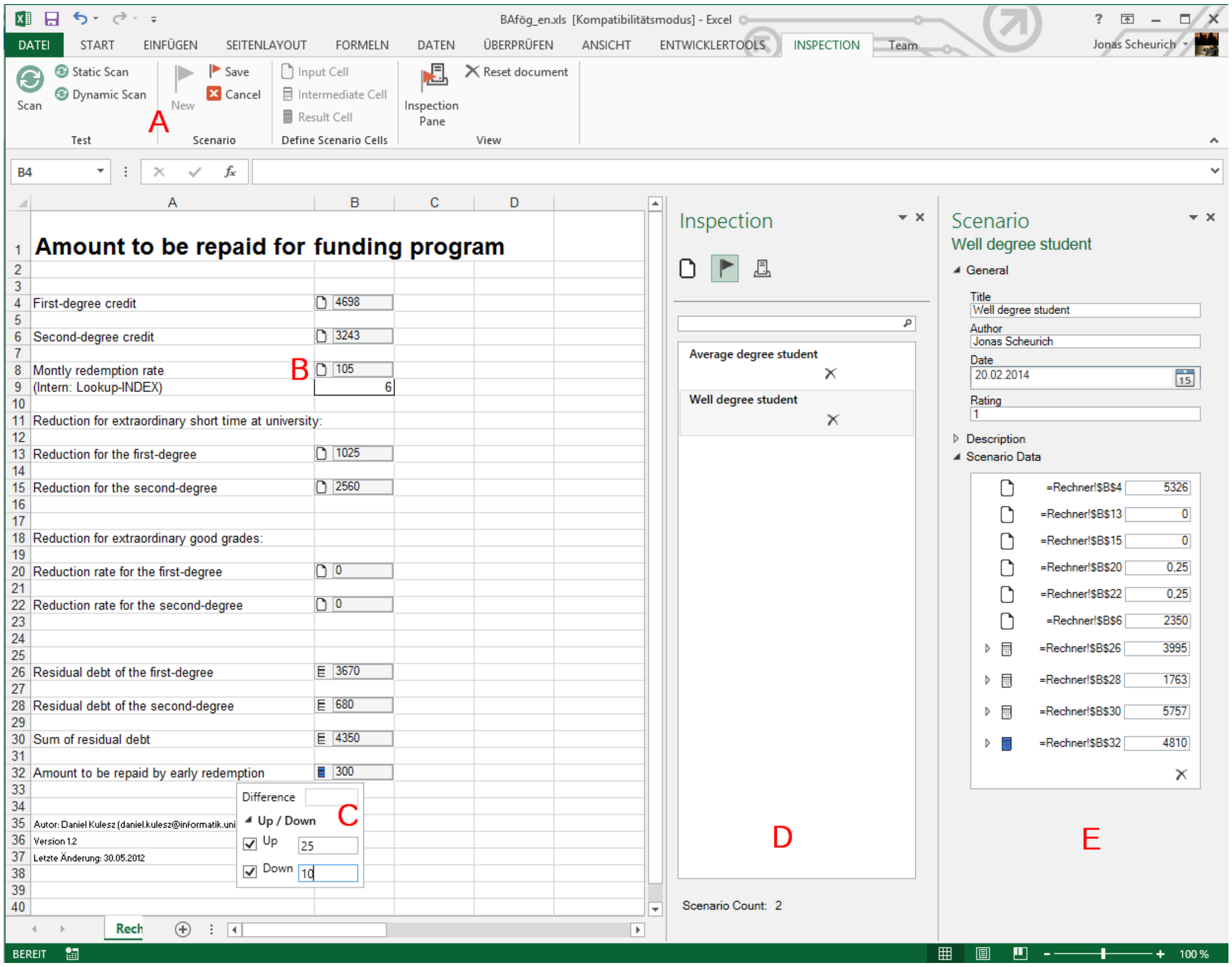
subject to frequent change. Many structural changes like the insertion of new or the removal of old columns or rows would certainly break all test scenarios if SIFEI stored cell markings as absolute references (e.g., "F3" or "E15"). To avoid this problem, SIFEI stores cell markings as hidden names. Therefore, test scenarios can survive many "typical" changes to a spreadsheet before having to be adapted.

2) *Test Scenario Editor:* The scenario editor being part of SIFEI implements our testing approach. It requires classifying important cells of the spreadsheet and filling these with input and expected output values. To demonstrate the editor, we use an example spreadsheet that calculates the amount of funds to be repaid for a scholarship (Figure 2). The spreadsheet is composed of 7 input cells, 3 intermediate calculation cells and one output cell. The formulas are very simple, with the exception of the formula in the output cell which uses VLOOKUP against a 6x51 grid of data cells in the second worksheet.

For classifying the cells, the user selects one or multiple cells and hits a button for marking the cell type in the ribbon bar of SIFEI (Figure 2, A). The button stays highlighted to indicate the defined cell type. Optionally, the side pane lists all cells that have already been marked (not shown in Figure 2).

The user enters the scenario creation mode by using a button in the ribbon menu (Figure 2, A). In scenario creation mode, all previously marked input cells are overlaid by a graphical control hiding the value that the cell was previously holding (Figure 2, B). The control consists of a symbol indicating the type of cell and a text box allowing the user to enter the values. For output cells, an additional context menu (Figure 2, C) allows the user to enter upwards and downwards deviation intervals as well. To provide a better overview, all created test scenarios are also shown in the scenario pane (Figure 2, D). Additionally, a detailed scenario view (Figure 2, E) allows the user to edit all values of one scenario. (Please note that the values for the scenario "well degree student" in Figure 2, E do not match the values shown in Figure 2, B because the latter values are part of a new and yet unnamed scenario that has not been saved yet.) SIFEI uses a low-intrusive way to keep scenarios in the spreadsheet files to enable the reuse of scenarios in future sessions.

After the user hits the 'scan' button in the ribbon menu, SIFEI passes all specified test scenarios to SIF. Then, for each defined scenario, SIF instantiates a new copy of the spreadsheet, fills the input cells with the data specified in the scenarios, and calculates the output values. Next, for each



	A	B
1	**Amount to be repaid for funding program**	
4	First-degree credit	4698
6	Second-degree credit	3243
8	Montly redemption rate	105
9	(Intern: Lookup-INDEX)	6
11	Reduction for extraordinary short time at university:	
13	Reduction for the first-degree	1025
15	Reduction for the second-degree	2560
18	Reduction for extraordinary good grades:	
20	Reduction rate for the first-degree	0
22	Reduction rate for the second-degree	0
26	Residual debt of the first-degree	3670
28	Residual debt of the second-degree	680
30	Sum of residual debt	4350
32	Amount to be repaid by early redemption	300
35	Autor: Daniel Kulesz (daniel.kulesz@informatik.uni	
36	Version 1.2	
37	Letzte Änderung: 30.05.2012	

Difference
Up / Down
Up: 25
Down: 10

Inspection
Average degree student
Well degree student
Scenario Count: 2

Scenario
Well degree student
General
Title: Well degree student
Author: Jonas Scheurich
Date: 20.02.2014
Rating: 1
Description
Scenario Data

=Rechner!$B$4	5326
=Rechner!$B$13	0
=Rechner!$B$15	0
=Rechner!$B$20	0,25
=Rechner!$B$22	0,25
=Rechner!$B$6	2350
=Rechner!$B$26	3995
=Rechner!$B$28	1763
=Rechner!$B$30	5757
=Rechner!$B$32	4810

Fig. 2. The SIFEI scenario editor.

intermediate and result cell, it compares the actual output values with the expected output values, taking into account the acceptable ranges defined in the scenario. Finally, for all deviations outside of acceptable ranges, SIF computes a report with findings and sends it back to SIFEI. The findings are visualized by SIFEI as described above.

V. Evaluation

We conducted two user studies to investigate the usability and usefulness of SIFEI. While the first study can be considered as a formative study to detect and fix usability issues of the tool, the second study investigates the testing approach in detail. Since issues raised in the first evaluation were fixed in the tool and this paper already describes the improved version of SIFEI, we concentrate on reporting findings of the second user study. This study focuses on evaluating the understandability of our testing approach and the applicability of SIFEI to support it. It was piloted by two participants and fine-tuned regarding phrasing and task design before running the actual study.

A. Setup

The study was designed as a "learn–apply–reflect" exercise: In the "learn part", participants were given a textual description that explained our cell terminology, the activities of marking cells, defining scenarios, executing inspections, and understanding findings by a tiny sandbox example. The complete description was only four A4 pages long and did not provide any direct help for interacting with SIFEI.

In the following "apply part", our participants were asked to carry out four tasks with SIFEI. For each task, participants were provided with a task description on one page and a simple spreadsheet in which they were supposed to carry out their activities. Thus, the impact from solving previous tasks correctly or incorrectly was limited for the subsequent tasks. Very briefly, the tasks were:

Task 1: Mark cell types

Task 2: Create a new test scenario

Task 3: Execute an inspection and interpret the findings

Task 4: Find a failure and specify a test scenario for it

For each task the participants were also provided with “help cards” (one single A4 paper each). These help cards were provided turned over and the participants were asked to use them only if they got stuck.

Finally, in the “reflect part” the participants were asked questions about their perception of our approach, SIFEI and their background regarding spreadsheets. After completing the experiment (including filling out the questionnaire), we also discussed our observations with the participants informally. In these informal discussions, we asked the participants which things seemed intuitive to them from the beginning and which things confused them.

B. Environment and Participants

The experiments were held one-by-one and on-site in a quiet room. We used a standard desktop computer for running SIF/SIFEI. The machine was connected to a 24-inch WUXGA display and standard input peripherals (keyboard and mouse). We told every participant that the experiment will go for a maximum of 50 minutes to provide enough time to complete the questionnaire in the last part. We did this in order to stay within the targeted one-hour time frame.

We observed the participants during the experiments by live-streaming their desktops and took notes on how they interacted with the tool. We paid special attention to how long they took for solving which task, pitfalls they encountered when using SIFEI, and when they used the provided help cards.

We only accepted participants with no profound knowledge of information science or software technology. We excluded computer scientists because they typically have a different background than “casual” spreadsheet users, approach problems differently, and, thus, are not representative for the targeted user group of our approach. We succeeded in attracting eight participants: Three aeronautics students, two physics students, one technician, one office secretary, one mechanical engineer, and one building physics engineer.

C. Observations

The results from the questionnaire are provided in Table 1. Below, we want to discuss them together with our observations of the four tasks:

Task 1: The results from the questionnaire indicate that all participants support the statement that they perceived marking input, intermediate, and output cells as easy. Our observations support these perceptions.

Task 2: The majority of our participants stated in the questionnaire that they had problems fully understanding the concept of test scenarios, although they perceived our solution for creating new scenarios to be acceptable. But from the discussions in the informal interviews, we gained the impression that most participants thought they understood the concept before trying to apply it with SIFEI. Apart from several minor usability issues, the problem we observed in almost every experiment was that, when trying to create a new scenario, the participants directly started entering data in the input cells and assumed that the ‘new’ button would create a new scenario using these values—but to their surprise, the values were gone (hidden) when entering the scenario editor, so they had to re-enter them.

Q1	Gender			
	female	male		
	2	6		
Q2	Age group			
	18-24 years	25-30 years	31-45 years	>45 years
	4	2	1	1
Q3	Education/studies completed			
	yes	no		
	6	2		
Q4	The phrasing of the tasks was understandable.			
	Fully agree			Not agree at all
	1	6	1	0
Q5	Marking cells as input cells or output cells was easy.			
	Fully agree			Not agree at all
	6	2	0	0
Q6	The concept of test scenarios was easy to understand.			
	Fully agree			Not agree at all
	1	4	3	0
Q7	The creation of test scenarios is solved well.			
	Fully agree			Not agree at all
	0	7	1	0
Q8	I believe that failures can be detected easier with SIFEI.			
	Fully agree			Not agree at all
	2	6	0	0
Q9	I would use SIFEI to inspect other spreadsheets as well.			
	Fully agree			Not agree at all
	1	3	3	1

TABLE I
RESULTS OF THE QUESTIONNAIRE (TRANSLATED FROM GERMAN).

Task 3: We did not ask for feedback on this task in the questionnaire, but observed that most participants had no problems in executing the inspection and interpreting the results correctly. The few problems detected were either related to bugs in the implementation of SIFEI or the participants not hitting the ‘scan’ button. The latter issue could be remedied in all cases by instructing our participants to make use of the provided help cards (some participants seemed to be a bit too shy to do that).

Task 4: The approaches that the participants took for solving this task were of particular interest to us, as we wanted to see whether they learned the lessons taught in the previous tasks. As we expected, the majority of participants (6 of 8) solved the task as follows: they tried various combinations for values in the input cells, changing them until they observed a failure. Then, they correctly defined the cell types, created a new scenario, executed it and received a finding. Two of our participants tried to find the failure by directly specifying test scenarios and executing them until they got a finding. Interestingly, most of our participants stated in the questionnaire that they believed that finding failures with SIFEI is easier than trying without.

D. Lessons Learned

After reflecting on the observations from Task 2, we considered planning the implementation of a feature that allows users to import data present in the input cells of the spreadsheet to a new test scenario—but the downside would be that the specification of expected results could be biased because users could have already seen the computed results in the spreadsheet and simply put them in as expected values.

Especially from the observations of our participants during Task 4, we learned that the approach intuitively taken by the majority of our participants indicates that our testing approach is probably better suited for documenting known failures than for finding new ones—at least for very simple spreadsheets like the ones we used in our evaluation.

Yet, the participants principally understood our approach after practicing it for less than fifty minutes. This is consistent with the insights gained from the informal discussions: Seven participants stated that they were confident to have understood our approach, but they were not yet convinced of its benefits. In all of these informal discussions, we explained that our approach provides a regression testing capability that might come in handy when spreadsheets are shared between users. After listening to these explanations, all of our participants stated that they now recognized the major benefit of our approach. However, please note that this is not reflected in the results of Q9 of the questionnaire because the interviews were held after the experiment.

E. Threats to Validity

From our controlled experiment with the small sample of eight participants, it is not sound to derive quantitative statements about the actual feasibility of both our testing approach and the tool support provided by SIFEI.

Certainly, the combined evaluation of both the approach and the tool in one experiment poses additional threats regarding the conclusions that can be drawn about the single parts—but since the proposed approach is not verifiable without tool support, an isolated evaluation seems even more challenging.

VI. Related work

We did not find related work that consistently integrates multiple diagnosis techniques into one spreadsheet environment, but there are publications concerning visualization of findings and partially automated testing approaches for spreadsheets. In the following we describe these works and show how they compare with our contributions.

Jannach et al. reviewed the literature on diagnosis techniques for spreadsheets [8] and propose a slightly different classification than the one we proposed in the introduction. Apart from the different terminology, they also mention "visualization-based approaches" (e.g., Breviz [9]) that visualize spreadsheets in a different way (e.g., using arrows that indicate cell dependencies) to aid users in detecting anomalies more easily. We accounted them as manual approaches and did not take them into closer consideration because they do not detect anomalies themselves and, thus, do not visualize any findings. However, some of the visualization techniques they employ might be regarded as interesting starting points for visualizing the context for particular findings (which SIFEI currently lacks).

Diagnosed findings in spreadsheets need to be communicated to the user. Only referencing the cell in a text output, however, is not a user-friendly presentation. Instead, visualizing the finding at the affected cells within the spreadsheet provides a better solution. A first step in this direction is coloring defective cells [10] or using a color scale for encoding severity [11], [12]. But since such colorings only act as markers, additional information is necessary to understand a finding. Users might retrieve it on demand using tooltip or pop-up dialogs [13], [11], [14]. Lists attached to the spreadsheet may provide an additional overview of all diagnosed findings [15].

A number of commercial static analysis tools also provides integration in spreadsheet environments. Kulesz and Ostberg [16] reviewed a number of practical challenges in such static analysis tools for spreadsheets and compared them to static analysis tools for traditional software. Some of these challenges—especially non-intrusiveness, presentation of findings, handling of false-positives, and the understandability of the inspection carried out by the tool—can surely be beneficial for tools that support partially automated detection approaches as well. Our tool SIFEI tries to incorporate the insights from the review into the design of its user interaction concept.

Apart from visualizing individual findings, Hermans et al. promote the use of so-called "risk maps" to communicate the severity of findings to users [11]. These risk maps are not to be confused with risk matrices but behave rather like simple heat maps. To draw them, Hermans et al. first remove all colors from the spreadsheet and then color the background of cells depending on the severity of the finding with three different colors: red, orange, and yellow. Compared with our approach to use an indicator icon, this is more intrusive because the original colors are removed.

One of the probably most prominent partially automated diagnostic approaches is WYSIWYT (*What You See Is What You Test*). Initially proposed as a generic approach for testing visual programs [17], it later was fine-tuned towards the spreadsheet-like programming environment Forms/3 [18], [5]. WYSIWYT is an incremental and interactive approach which basically works as follows: Values in input cells are populated by either the user or a random generator. Next, the spreadsheet is executed and users are asked to mark the values in intermediate and output cells as being correct or wrong, getting feedback about the progress of "testedness" during this activity. Our approach is different with respect to the unprejudiced specification of expected output values: Unlike WYSIWYT, we separate the specification of test scenarios from the current state of the spreadsheet. We intentionally do not present results to the users asking them to only check them for correctness to avoid the problem of overconfidence: Otherwise, users could tend to inspect the formulas and, e.g., assume that a result is correct if the cell only contains a sum formula referring to the right cells—while, in fact, one of these cells might be formated as text and not accounted into the sum. Therefore, we find it worthwhile making users think about expected output values without seeing the computed result beforehand.

Ayalew [19], [20] proposes an approach called "interval testing" where users specify plausible intervals for values in all numeric cells (i.e., input cells, intermediate cells, and output cells). Using symbolic execution based on the dependency chain between these cells, an interval for the minimum and maximum possible values for each intermediate and result cell is computed. Then, these bounding intervals are compared with the intervals specified by the user and mismatches reported as findings that can be traced using the dataflow information [21]. Compared with our approach which asks the user to specify multiple test scenarios, the specification of global boundaries is sufficient for interval testing. But this is a trade-off as the chance to detect anomalies is smaller if only boundaries of values are concerned. Ayalew argues that one of the reasons that motivated him in choosing this trade-off was the spreadsheet users' lack of patience and expertise to run a lengthy suite of test cases. We try to mitigate this issue by providing the ability to run the test suite automatically once the test scenarios are specified.

Regression testing is a standard diagnosis technique for professional software engineers. Unit testing frameworks are integral parts of many IDEs. For instance, JUnit as integrated in Eclipse shows a list of defects and connects these to the test cases that caused them. Pinpointing the defect location within the productive code is more difficult than in spreadsheets but can be estimated using, for instance, spectrum-based fault localization [22], [19], [23]: a computed defect probability for each statement is visualized in the background of the code. Combining this approach with a back-in time debugger further eases the fault localization process [24]. Other information that might help debugging has also been visually integrated into source code, such as software metrics [25] or runtime consumption of methods [26].

Debugging is also the primary goal of the EXQUISITE approach proposed by Jannach et al. [15]. At first glance, EXQUISITE has several similarities with our approach (e.g., they also visualize findings individually when the user highlights them). However, its interaction design follows the WYSIWYT approach and allows users to flag the (in)correctness of single values in cells independent of test cases. They extend the WYSIWYT approach and allow users to flag complete test cases (consisting of input and output values) as correct or incorrect. By providing values for incorrect cells, the user can also quickly transform negative test cases into positive ones. Although the specification of test cases in EXQUISITE is also carried out in a different view ("debug mode"), it does not hide the values in the spreadsheet and, thus, might suffer from the overconfidence problem as well.

A fundamental difference between our approach and EXQUISITE as well as of all the other approaches discussed in this section is the way how the meaning of cells (input, intermediate, and output cells) is retrieved. We have the only approach that does not retrieve the meaning automatically without the help of the user. Kulesz [27] argues that automatic retrieval of cell meanings is problematic because there are cases where automatic detection would fail. One example is when spreadsheets contain so-called "test formulas" [28] that are formulas meant to check output cells—automatic approaches would regard them as output values, while in fact the user would regard the intermediate cells they refer to as output cells. Another example are input cells that the user regards as constants and, thus, would never expect them to change across different test cases. Thus, such misconceptions about cell meanings might become apparent when explicitly classifying cell meanings.

Hermans [28] evaluated 4000 spreadsheets from the EUSES corpus [29] (a corpus of "random" spreadsheets which were publicly accessible on the Internet in 2005). She found that 8.8% of all unique formulas in this spreadsheet sample were actually unique test formulas. This means that a significant number of cells that would be typically detected as output cells actually contain formulas that only serve the purpose of error handling. Hermans proposes to help users to increase the quality of these formulas by increasing the test coverage of these formulas and proposes the "Expector" approach to achieve this goal. Although we see the benefits in increasing test coverage, we do not want to support users in practicing this style of defensive programming. Therefore, we do not store our user-specified test scenarios in locations that are visible during normal use. Instead, we clearly separate the production code (formulas in the spreadsheet) from the testing code (cell markings and test scenarios).

VII. Conclusion and future work

In this paper, we proposed a partially automated testing approach for detecting failures in spreadsheets, where the output of the spreadsheet does not match the user's expectations. The approach differs in a number of key concepts from existing approaches. It propagates a new user interaction design that allows users with no previous knowledge about testing to create test scenarios and interpret findings from their execution. Our tool SIFEI integrates the approach with the popular spreadsheet execution environment Microsoft Excel.

The conducted qualitative evaluation indicates that the testing approach is understandable in less than 50 minutes even for casual spreadsheet users. Furthermore, the results show that, with the help of SIFEI, our proposed testing approach can be successfully applied in practice—at least in the tested scenario for simple spreadsheets.

Nevertheless, one important question yet neglected is the motivation of end-users to use our approach or spreadsheet testing in general. Just because end users would succeed in test case specification, it does not mean that they will also create them from their own initiative. Good motivation strategies (e.g., [30]) are needed to convince end-users to carry out activities that should have positive long-term effects but do not show immediate benefits for the tasks users are solving using spreadsheets.

We learned from traditional software engineering that the most effective strategy for detecting anomalies is the combination of different techniques (i.e., static analysis, unit testing, system testing, manual reviews). We are convinced that this holds true for spreadsheet as well and, thus, it is worthwhile to further exploit the path of integrating multiple anomaly diagnosis techniques for spreadsheets.

ACKNOWLEDGMENT

This work would not have been possible without the help of the students who actively contributed to the development of SIF and SIFEI. We would like to thank especially Ehssan Doust, who conducted the visualization part of SIFEI that states a significant contribution for this paper. Furthermore, we would like to thank all participants of our qualitative studies.

REFERENCES

[1] R. R. Panko, "Spreadsheet errors: What we know. what we think we can do," *Proceedings of the 2000 EuSpRIG Conference*, 2000.

[2] ——, "Two experiments in reducing overconfidence in spreadsheet development," *Journal of Organizational and End User Computing (JOEUC)*, vol. 19, no. 1, pp. 1–23, 2007.

[3] J. Cunha, J. P. Fernandes, P. Martins, J. Mendes, and J. Saraiva, "Smellsheet detective: A tool for detecting bad smells in spreadsheets," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 2012, pp. 243–244.

[4] R. Abraham and M. Erwig, "Autotest: A tool for automatic test case generation in spreadsheets," in *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 2006, pp. 43–50.

[5] M. Fisher II, G. Rothermel, D. Brown, M. Cao, C. Cook, and M. Burnett, "Integrating automated test generation into the WYSIWYT spreadsheet testing methodology," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 2, pp. 150–194, 2006.

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

[7] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[8] D. Jannach, T. Schmitz, B. Hofer, and F. Wotawa, "Avoiding, finding and fixing spreadsheet errors–a survey of automated approaches for spreadsheet QA," *Journal of Systems and Software (to appear)*, 2014.

[9] F. Hermans, "Analyzing and visualizing spreadsheets," Ph.D. dissertation, PhD thesis, Software Engineering Research Group, Delft University of Technology, Netherlands, 2012.

[10] R. Abraham and M. Erwig, "Header and unit inference for spreadsheets through spatial analyses," in *Visual Languages and Human-Centric Computing (VL/HCC), 2004 IEEE Symposium on*. IEEE, 2004, pp. 165–172.

[11] F. Hermans, M. Pinzger, and A. Deursen, "Detecting and refactoring code smells in spreadsheet formulas," *Empirical Software Engineering*, pp. 1–27, 2014.

[12] J. Reichwein, G. Rothermel, and M. Burnett, "Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging," *ACM SIGPLAN Notices*, vol. 35, no. 1, pp. 25–38, 1999.

[13] R. Abraham and M. Erwig, "How to communicate unit error messages in spreadsheets," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[14] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main, "End-user software visualizations for fault localization," in *Proceedings of the 2003 ACM symposium on Software visualization (SoftVis)*. ACM, 2003, pp. 123–132.

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

[16] D. Kulesz and J.-P. Ostberg, "Practical Challenges with Spreadsheet Auditing Tools," in *Proceedings of the 2013 EuSpRIG Conference*, 2013, pp. 1–13.

[17] 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 *Software Engineering, 1998. Proceedings of the 1998 International Conference on*. IEEE, 1998, pp. 198–207.

[18] M. Burnett, A. Sheretov, B. Ren, and G. Rothermel, "Testing homogeneous spreadsheet grids with the 'what you see is what you test' methodology," *Software Engineering, IEEE Transactions on*, vol. 28, no. 6, pp. 576–594, 2002.

[19] Y. Ayalew, "Spreadsheet testing using interval analysis," Ph.D. dissertation, Klagenfurt University, 2001.

[20] ——, "A user-centered approach for testing spreadsheets," *International Journal of Computing and ICT Research*, vol. 1, no. 1, pp. 77–85, 2007.

[21] Y. Ayalew and R. Mittermeir, "Spreadsheet debugging," *Proceedings of the 2003 EuSpRIG Conference*, pp. 67–79, 2003.

[22] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. ACM, 2002, pp. 467–477.

[23] R. Abreu, A. Riboira, and F. Wotawa, "Constraint-based debugging of spreadsheets." in *Ibero-American Conference on Software Engineering*, 2012, pp. 1–14.

[24] M. Perscheid and R. Hirschfeld, "Follow the path: Debugging tools for test-driven fault navigation," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 446–449.

[25] M. Harward, W. Irwin, and N. Churcher, "In situ software visualisation," in *Proceedings of the 21st Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society, 2010, pp. 171–180.

[26] F. Beck, O. Moseler, S. Diehl, and G. D. Rey, "In situ understanding of performance bottlenecks through visually augmented code," in *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 63–72.

[27] D. Kulesz, "A spreadsheet cell-meaning model for testing," *accepted for publication at the Workshop on Software Engineering Methods in Spreadsheets*, 2014.

[28] F. Hermans, "Improving spreadsheet test practices," *Center for Advanced Studies on Collaborative Research, CASCON*, 2013.

[29] M. Fisher and G. Rothermel, "The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[30] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel, "Harnessing curiosity to increase correctness in end-user programming," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2003, pp. 305–312.