Toward Interactive Spreadsheet Debugging

Dietmar Jannach
TU Dortmund, Germany
dietmar.jannach@tu-dortmund.de

Thomas Schmitz
TU Dortmund, Germany
thomas.schmitz@tu-dortmund.de

Kostyantyn Shchekotykhin
University Klagenfurt, Austria
kostya@ifit.uni-klu.ac.at

ABSTRACT

Spreadsheet applications are often developed in a comparably unstructured process without rigorous quality assurance mechanisms. Faults in spreadsheets are therefore common and finding the true causes of an unexpected calculation outcome can be tedious already for small spreadsheets. The goal of the EXQUISITE project is to provide spreadsheet developers with better tool support for fault identification. EXQUISITE is based on an algorithmic debugging approach relying on the principles of Model-Based Diagnosis and is designed as a plug-in to MS Excel. In this paper, we give an overview of the project, outline open challenges, and sketch different approaches for the interactive minimization of the set of fault candidates.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Spreadsheets; D.2.8 [**Software Engineering**]: Testing and Debugging

1. INTRODUCTION

Spreadsheet applications are mostly developed in an unstructured, ad-hoc process without detailed domain analysis, principled design or in-depth testing. As a result, spreadsheets might often be of limited quality and contain faults, which is particularly problematic when they are used as decision making aids. Over the last years, researchers have proposed a number of ways of transferring principles, practices and techniques of software engineering to the spreadsheet domain, including modeling approaches, better test support, refactoring, or techniques for problem visualization, fault localization, and repair [2, 5, 9, 13, 15].

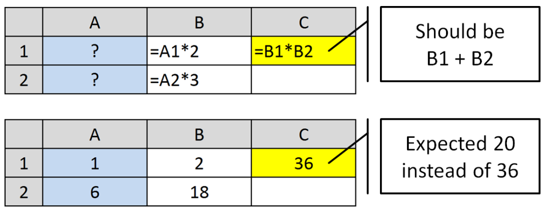
The EXQUISITE project [12] continues these lines of research and proposes a constraint-based approach for algorithmic spreadsheet debugging. Technically, the main idea is to translate the spreadsheet under investigation as well as user-specified test cases into a Constraint Satisfaction Problem (CSP) and then use Model-Based Diagnosis (MBD) [14] to find the diagnosis candidates. In terms of a CSP, each candidate is a set of constraints that have to be modified to correct a failure. In our previous works, we demonstrated the general feasibility of the approach and presented details of an MS Excel plug-in, which allows the user to interactively specify test cases, run the diagnosis process and then explore the possible candidates identified by our algorithm [12].

Using constraint reasoning and diagnosis approaches for spreadsheet debugging – partially in combination with other techniques – was also considered in [3, 4, 11]. While these techniques showed promising results in helping users to locate faults in spreadsheets, a number of challenges remain. In this paper, we address the question of how the end user can be better supported in situations when many diagnosis candidates are returned by the reasoning engine. We will sketch different interactive candidate discrimination approaches in which the user is queried by the system about the correctness of individual cells' values and formulas.

2. DIAGNOSING SPREADSHEETS

Algorithmic Approach.

In [14], Reiter proposed a domain-independent and logic-based characterization of the MBD problem. A diagnosable system comprises a set of interconnected components COMPS, each of which can possibly fail. A system description SD specifies how components behave when they work correctly, i.e., given some inputs, the definitions in SD and COMPS determine the expected outputs. In case the expected outputs deviate from what is actually observed, the diagnosis problem consists of identifying a subset of COMPS, which, if assumed faulty, explains the observations.



	A	B	C
1	?	=A1*2	=B1*B2
2	?	=A2*3	

	A	B	C
1	1	2	36
2	6	18	

Should be B1 + B2
Expected 20 instead of 36

Figure 1: A spreadsheet with an unexpected output

The main idea can be transferred to the spreadsheet domain as follows [12]. In the example shown in Figure 1, the intended formula in cell C2 should be an addition, but the developer made a typo. When testing the spreadsheet with the inputs {A1=1, A2=6} and the expected output {C1=20}, the user notices an unexpected output (36) in $C1$. MBD reasoning now aims to find minimal subsets of the possibly faulty components – in our case the cells with formulas – which can explain the observed discrepancy. A closer investigation of the problem reveals that only two minimal explanations exist in our example if we only allow integer values: "C1 is faulty" and "B2 is faulty". The formula in cell B1 alone cannot be the sole cause of the problem with B2 and C1 being correct as 18 is not a factor of the expected value 20, i.e., there is no solution to the equation

$B1 \cdot 18 = 20, B1 \in \mathbb{N}$. Note that we assume that the constant values in the spreadsheet are correct. However, our approach can be easily extended to deal with erroneous constants.

In [12], we describe a plug-in component for MS Excel, which relies on an enhanced and parallelized version of this diagnostic procedure, additional dependency-based search space pruning, and a technique for fast conflict detection.

We have evaluated our approach in two ways. (A) We analyzed the required running times using a number of spreadsheets in which we injected faults (mutations). The results showed that our method can find the diagnoses for many small- and mid-sized spreadsheets containing about 100 formulas within a few seconds. (B) We conducted a user study in the form of an error detection exercise involving 24 subjects. The results showed that participants who used the Exquisite tool were both more effective and efficient than those who only relied on MS Excel's standard fault localization mechanisms. A post-experiment questionnaire indicated that both groups would appreciate better tool support for fault localization in commercial tools [12].

The Problem of Discriminating Between Diagnoses.

While we see our results so far to be promising, an open issue is that the set of diagnosis candidates can be large. Finding the true cause of the error within a larger set of possible explanations can be tedious and, finally, make the approach impractical when a user has to inspect too many alternatives. Since this is a general problem of MBD, the question of how to help the user to better discriminate between the candidates and focus on the most probable ones was in the focus of several works from the early days of MBD research [7]. In the next section, we will propose two possible remedies for this problem in the spreadsheet domain.

3. TOWARD INTERACTIVE DEBUGGING

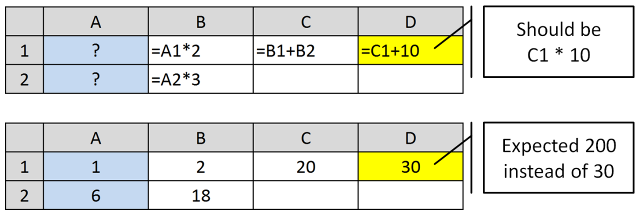
Early works in MBD research were dealing with fault diagnosis of electrical circuits. In this domain, an engineer can make additional measurements, e.g., of the voltage at certain nodes. These measurements can then help to reduce the set of candidates because some observations may rule out certain explanations for the observed behavior.

Each measurement however induces additional costs or effort from the user. One goal of past research was thus to automatically determine "good" measurement points, i.e., those which help to narrow down the candidate space fast and thus minimize the number of required measurements. In [7], for example, an approach based on information theory was proposed where the possible measurement points were ranked according to the expected information gain.

3.1 Example

Figure 2 exemplifies how additional measurements (inputs by the user) can help us to find the cause of a fault. The example is based on the one from Figure 1. The user has corrected the formula in C1 and added a formula in D1 that should multiply the value of C1 by 10. Again, a typo was made and the observed result in D1 is 30 instead of the expected value of 200 for the input values {A1=1, A2=6}.

Given this unluckily chosen test case, MBD returns four possible single-element candidates {B1}, {B2}, {C1}, and {D1}, i.e., every formula could be the cause. To narrow down this set, we could query the user about the correctness of the intermediate results in the cells B1, B2, and C1.



	A	B	C	D
1	?	=A1*2	=B1+B2	=C1+10
2	?	=A2*3		

Should be C1 * 10

	A	B	C	D
1	1	2	20	30
2	6	18		

Expected 200 instead of 30

Figure 2: Another faulty spreadsheet

If we ask the user for a correct value of B1, then the user will answer "2". Based on that information, B1 can be ruled out as a diagnosis and the problem must be somewhere else. However, if we had asked the user for the correct value of C1, the user would have answered "20" and we could immediately infer that {D1} remains as the only possible diagnosis.

The question arises how we can automatically determine which questions we should ask to the user. Next, we sketch two possible strategies for interactive querying.

3.2 Querying for Cell Values

The first approach (Algorithm 1) is based on interactively asking the user about the correct values of intermediate cells as done in the example[1].

Algorithm 1: Querying cell values

```
Input: A faulty spreadsheet $\mathcal{P}$, a test case $\mathcal{T}$
$\mathcal{S}$ = Diagnoses for $\mathcal{P}$ given $\mathcal{T}$
while $|\mathcal{S}| > 1$ do
    foreach intermediate cell $c \in \mathcal{P}$ not asked so far do
        $val$ = computed value of $c$ given $\mathcal{T}$
        count($c$) = 0
        foreach Diagnosis $d \in \mathcal{S}$ do
            $CSP' = CSP$ of $\mathcal{P}$ given $\mathcal{T} \setminus$ Constraints($d$)
            if $CSP' \cup \{c = val\}$ has a solution then
                inc(count($c$))
    Query user for expected value $v$ of the cell $c$ where
        count($c$) is minimal
    $\mathcal{T} = \mathcal{T} \cup \{c = v\}$
    $\mathcal{S}$ = Diagnoses for $\mathcal{P}$ given $\mathcal{T}$
```

The goal of the algorithm is to minimize the number of required interactions. Therefore, as long as there is more than one diagnosis, we determine which question would help us most to reduce the set of remaining diagnoses. To do so, we check for each possible question (intermediate cell c), how many diagnoses would remain if we knew that the cell value val is correct given the test case $\mathcal{T}$. Since every diagnosis candidate d corresponds to a relaxed version CSP' of the original CSP, where the latter is a translation of the spreadsheet $\mathcal{P}$ and the test case $\mathcal{T}$, we check if CSP' together with the assignment $\{c = val\}$ has a solution. Next, we ask the user for the correct value of the cell for which the smallest number of remaining diagnoses was observed. The user-provided value is then added to the set of values known to be correct for $\mathcal{T}$ and the process is repeated.

To test the approach, we used a number of spreadsheets containing faults from [12], measured how many interactions

[1]Actually, a user-provided range restriction for C1 ($15 <$ C1 < 25) would have been sufficient in the example.

Scenario	#C	#F	#D	∅Rand	Min
Sales forecast	143	1	89	46.7	11
Hospital form	38	1	15	7.6	5
Revenue calc.	110	3	200	11.8	9
Example Fig. 3	170	1	85	50.3	12

Table 1: Results for querying cell values.

it requires to isolate the single correct diagnosis using Algorithm 1 and compared it to a random measurement strategy.

The results given in Table 1 show that for the tested examples the number of required interactions can be measurably lowered compared to a random strategy. The sales forecast spreadsheet, for example, comprises 143 formulas (#C) and contains 1 fault (#F). Using our approach, only 11 cells (Min) have to be inspected by the user to find the diagnosis explaining a fault within 89 diagnosis candidates (#D). Repeated runs of the randomized strategy lead to more than 45 interactions (∅Rand) on average.

As our preliminary evaluation shows, the developed heuristic decreases the number of user interaction required to find the correct diagnosis. In our future work, we will also consider other heuristics. Note however that there are also problem settings in which all possible queries lead to the same reduction of the candidate space. Nonetheless, as the approach shows to be helpful at least in some cases, we plan to explore the following extensions in the future.

- Instead of asking for expected cell values we can ask for the correctness of individual calculated values or for range specifications. This requires less effort by the user but does not guarantee that one single candidate can be isolated.

- Additional test cases can help to rule out some candidates. Thus, we plan to explore techniques for automated test-case generation. As spreadsheets often consist of multiple blocks of calculations which only have few links to other parts of the program, one technique could be to generate test cases for such smaller fragments, which are easier to validate for the user.

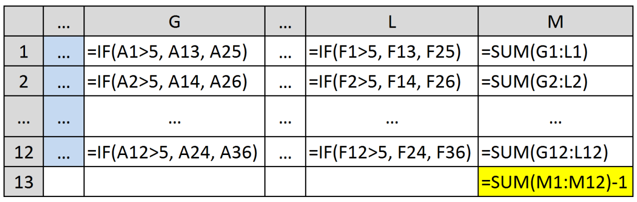
3.3 Querying for Formula Correctness

Calculating expected values for intermediate cells can be difficult for users as they have to consider also the cells preceding the one under investigation. Thus, we propose additional strategies in which we ask for the correctness of individual formulas. Answering such queries can in the best case be done by inspecting only one particular formula.

1. We can query the user about the elements of the most probable diagnoses as done in [16], e.g., by limiting the search depth and by estimating fault probabilities.

2. In case of multiple-fault diagnoses, we can ask the user to inspect those formulas first that appear in the most diagnoses. If one cell appears in all diagnoses, it must definitely contain an error.

3. After having queried the user about the correctness of one particular formula, we can search for copy-equivalent formulas and ask the user to confirm the correctness of these formulas.



	...	G	...	L	M
1	...	=IF(A1>5, A13, A25)	...	=IF(F1>5, F13, F25)	=SUM(G1:L1)
2	...	=IF(A2>5, A14, A26)	...	=IF(F2>5, F14, F26)	=SUM(G2:L2)
...	...	...	...	...	...
12	...	=IF(A12>5, A24, A36)	...	=IF(F12>5, F24, F36)	=SUM(G12:L12)
13					=SUM(M1:M12)-1

Figure 3: A small extract of a faulty spreadsheet with structurally identical formulas

The rationale of this last strategy, which we will now discuss in more detail, is that in many real-world spreadsheets, structurally identical formulas exist in neighboring cells, which, for example, perform a row-wise aggregation of cell values. Such repetitive structures are one of the major reasons that the number of diagnosis candidates grows quickly. Thus, when the user has inspected one formula, we can ask him if the given answer also applies to all copy-equivalent formulas, which we can automatically detect.

In the example spreadsheet shown in Figure 3 the user made a mistake in cell M13 entering a minus instead of the intended plus symbol. A basic MBD method would in the worst case and depending on the test cases return every single formula as equally ranked diagnosis candidates. When applying the value-based query strategy of Section 3.2, the user would be asked to give feedback on the values of M1 to M12, which however requires a lot of manual calculations.

With the techniques proposed in this section, the formulas of the spreadsheet would first be ranked based on their fault probability. Let us assume that our heuristics say that users most probably make mistakes when writing IF-statements. In addition, the formula M13 is syntactically more complex as those in M1 to M12 and thus more probable to be faulty.

Based on this ranking, we would, for example, ask the user to inspect the formula of G1 first. Given the feedback that the formula is correct, we can ask the user to check the copy-equivalent formulas of G1 to L12. This task, however, can be very easily done by the user by navigating through these cells and by checking if the formulas properly reflect the intended semantics, i.e., that the formulas were copied correctly. After that, the user is asked to inspect the formula M13 according to the heuristic which is actually the faulty one. In this example, we thus only needed 3 user interactions to find the cause of the error. In a random strategy, the user would have to inspect up to half of the formulas in average depending on the test case. The evaluation of the described techniques is part of our ongoing work.

3.4 User Acceptance Issues

Independent of the chosen strategy, user studies have to be performed to assess which kinds of user input one can realistically expect, e.g., for which problem scenarios the user should be able to provide expected (ranges of) values for intermediate cells. In addition, the spreadsheet inspection exercise conducted in [12] indicates that users follow different fault localization strategies: some users for example start from the inputs whereas others begin at the “result cells”. Any interactive querying strategy should therefore be carefully designed and assessed with real users. As a part of a future work we furthermore plan to develop heuristics to select one of several possible debugging techniques depending on the users problem identification strategy.

4. ADDITIONAL CHALLENGES

Other open issues in the context of MBD-based debugging that we plan to investigate in future work include the following aspects.

Probability-Based Ranking.

Another approach to discriminate between diagnoses is to try to rank the sometimes numerous candidates in a way that those considered to be the most probable ones are listed first. Typically, one would for example list diagnoses candidates of smaller cardinality first, assuming that single faults are more probable than double faults. In addition, we can use fault statistics from the literature for different types of faults to estimate the probability of each diagnosis. In the spreadsheet domain, we could also rely on indicators like formula complexity or other spreadsheet smells [10], the location of the cell within the spreadsheet's overall structure, results from Spectrum-Based Fault Localization [11], or the number of recent changes made to a formula. User studies in the form of spreadsheet construction exercises as done in [6] can help to identify or validate such heuristics.

Problem Encoding and Running Times.

For larger problem instances, the required running times for the diagnosis can exceed what is acceptable for interactive debugging. Faster commercial constraint solvers can alleviate this problem to some extent. However, also automated problem decomposition and dependency analysis methods represent a powerful means to be further explored to reduce the search complexity.

Another open issue is that in works relying on a CSP-encoding of the spreadsheets, e.g., [3, 11] and our work, the calculations are limited to integers, which is caused by the limited floating-point support of free constraint solvers. More research is required in this area, including the incorporation of alternative reasoning approaches like, e.g., linear optimization.

User Interface Design.

Finally, as spreadsheet developers are usually not programmers, the user interface (UI) design plays a central role and suitable UI metaphors and a corresponding non-technical terminology have to be developed. In Exquisite, we tried to leave the user as much as possible within the known MS Excel environment. Certain concepts like "test cases" are, however, not present in modern spreadsheet tools and require some learning effort from the developer. The recent work of [8] indicates that users are willing to spend some extra effort, e.g., in test case specification, to end up with more fault-free spreadsheets.

How the interaction mechanisms actually should be designed to be usable at least by experienced users, is largely open in our view. In previous spreadsheet testing and debugging approaches like [1] or [2], for example, additional input was required by the user. In-depth studies about the usability of these extensions to standard spreadsheet environments are quite rare.

5. SUMMARY

In this paper, we have discussed perspectives of constraint and model-based approaches for algorithmic spreadsheet debugging. Based on our insights obtained so far from the Exquisite project, we have identified a number of open challenges in the domain and outlined approaches for interactive spreadsheet debugging.

Acknowledgements

This work was supported by the EU through the programme "Europäischer Fonds für regionale Entwicklung - Investition in unsere Zukunft" under contract number 300251802.

6. REFERENCES

[1] R. Abraham and M. Erwig. AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets. In *Proceedings VL/HCC 2006*, pages 43–50, 2006.

[2] R. Abraham and M. Erwig. GoalDebug: A Spreadsheet Debugger for End Users. In *Proc. ICSE 2007*, pages 251–260, 2007.

[3] R. Abreu, A. Riboira, and F. Wotawa. Constraint-based Debugging of Spreadsheets. In *Proc. CIbSE 2012*, pages 1–14, 2012.

[4] S. Außerlechner, S. Fruhmann, W. Wieser, B. Hofer, R. Spörk, C. Mühlbacher, and F. Wotawa. The Right Choice Matters! SMT Solving Substantially Improves Model-Based Debugging of Spreadsheets. In *Proc. QSIC 2013*, pages 139–148, 2013.

[5] S. Badame and D. Dig. Refactoring meets Spreadsheet Formulas. In *Proc. ICSM 2012*, pages 399–409, 2012.

[6] P. S. Brown and J. D. Gould. An Experimental Study of People Creating Spreadsheets. *ACM TOIS*, 5(3):258–272, 1987.

[7] J. de Kleer and B. C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[8] F. Hermans. Improving Spreadsheet Test Practices. In *Proc. CASCON 2013*, pages 56–69, 2013.

[9] F. Hermans, M. Pinzger, and A. van Deursen. Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams. In *ICSE 2011*, pages 451–460, 2011.

[10] F. Hermans, M. Pinzger, and A. van Deursen. Detecting Code Smells in Spreadsheet Formulas. In *Proc. ICSM 2012*, pages 409–418, 2012.

[11] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner. On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets. In *Proc. FASE 2013*, pages 68–82, 2013.

[12] D. Jannach and T. Schmitz. Model-based diagnosis of spreadsheet programs - A constraint-based debugging approach. *Autom Softw Eng*, to appear, 2014.

[13] 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.

[14] R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[15] 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. ICSE 1998*, pages 198–207, 1998.

[16] K. Shchekotykhin, G. Friedrich, P. Fleiss, and P. Rodler. Interactive ontology debugging: Two query strategies for efficient fault localization. *Journal of Web Semantics*, 12-13:788–103, 2012.