Heuristics for the Automatic Identification of Irregularities in Spreadsheets

Markus Clermont
Software Quality Research Laboratory
University of Limerick
IRELAND
markus.clermont@ul.ie

ABSTRACT

Spreadsheet programs turned out to be the most popular end-user programming environment that has ever been released. Important decisions are based on the results of spreadsheet programs and the list of known errors with large impact is growing daily- although it surely is only the top of an iceberg.

One way out of the crisis might be the introduction of software engineering techniques into spreadsheet development. Suggestions for the improvement of spreadsheet development range back as far as into the late eighties, but none has been successful yet. We argue this is either because not enough effort is put into the roll-out of the technique to the users and, mainly, because they neglect the fact that spreadsheet programmers are end-users, not willing or not able to spend any time on learning software engineering methods. We found out that most end users are willing to verify their spreadsheets, but only view have the time and skills to do really systematic testing of spreadsheets.

We developed an approach to generate two orthogonal abstract representations of spreadsheet programs that are then displayed to the user by different visualisation techniques to support the auditing process. Usually, irregularities in the visualisation point out hot-spots on the spreadsheet with a high likelihood of erroneous formulas. In this paper we present new heuristics for identifying hot spots that are very efficient for large spreadsheet programs.

Categories and Subject Descriptors

H.4.1 [**Information Systems Applications**]: Office Automation-Spreadsheets; D.2.5 [**Software**]: Software EngineeringTesting and Debugging

General Terms

Algorithms

Keywords

Program Analysis, Spreadsheet Visualization, End User Programming

1. INTRODUCTION

It has been shown, that spreadsheets are used by a vast majority of people in the upper- and middle management of today's business world [18, 5]. Hence, it is no surprise that many important decisions are based on the results of spreadsheet programs.

For a software engineer a spreadsheet program is obviously software and thus should be developed obeying some systematic approach and then be carefully tested. For the typical spreadsheet user, who is not a software engineer but usually an expert in the application domain, a spreadsheet program is not considered as software. For them it is a tool for performing calculations and formatting their results. Spreadsheets are often considered as a word-processor for numbers, and not as the highly complex data flow program that they really are. Hence, it is not surprising that the end users shy away from software engineering approaches. And that there is a long list of well document spreadsheet-error horror stories, e.g. at the web-page of the European Spreadsheet Risk Interest Group [17].

Although there are already a couple of possible methods to either enforce a systematic development of spreadsheets, according to software engineering principles (see [4, 14, 22, 15] or to reduce the error rate of already existing spreadsheets, by testing (see [1, 23, 21]) or auditing (see [3, 24] and other commercially distributed tools), they are still not widely accepted. One reason for the poor acceptance of approaches that require a systematic development of spreadsheet programs is the nature of the spreadsheet as a prototyping tool itself. Another reason for the failure of many testing and auditing approaches is the sheer size of the spreadsheets that are common in industry. In [16] we report of a field study auditing the spreadsheets of a large international company. We examined 78 spreadsheets, with the average spreadsheet containing more than 2400 non-empty cells. Testing the whole spreadsheet, even with the support of current tools and techniques, still remains a tedious task.

These facts are not new, and it was found out earlier by Panko (see [20]) that checking a spreadsheet is a time consuming and expensive task. Thus, we argue similar to Butler [3], if time is scarce systematic auditing or testing should be limited to the crucial parts of spreadsheets that are most likely to be to be erroneous. However, it is not trivial to identify these parts in a quick and efficient way. There are some methods that operate on user assessment of the risk and the impact of an error in a certain region of the spreadsheet (see [3]), but they are subject to the auditors attitude and might not map to the actual erroneous areas of spreadsheet programs. The visualisation approach discussed in [9, 7, 6] offers already an approach to identify certain irregularities of the spreadsheet by means of an comprehensible abstract visualisation of the audited spread-

sheet that is always connected with the spreadsheet system's user interface that is familiar to the spreadsheet programmer [10, 8].

In this paper we want to introduce four new heuristics based on the technique mentioned above, to automatically discover hot-spots in spreadsheet programs where errors are likely to appear. Therefore, we will briefly introduce our abstraction technique in the next section and point out the heuristics to detect hot-spots known so far. In Section 3 we will introduce the four new heuristics, and finally we will shortly discuss the limits of our approach.

2. LOGICAL AREAS, SEMANTIC CLASSES AND DATA MODULES

Basically we developed two strategies to generate abstract representations of spreadsheet programs- the first one, semantic classes [9], is based on the cell's contents, i.e. the formulas, and their placement on the spreadsheet, whereas the second one, data modules [6], takes only cell dependencies into account. We will give only a very brief and informal introduction here, for a more detailed discussion we refer to the sources cited above or to [7].

2.1 Logical Areas

Semantic classes are an extension of the concept of logical areas that were first introduced in [2]. Logical areas are an attempt to construct an abstract presentation of a given spreadsheet program by considering only the cells contents and grouping cells based on the similarity of their contents into equivalence classes. In contrast to other techniques that group cells based on their contents, e.g. [24], there are two main differences:

1. The spatial location of cells on the spreadsheet is not taken into account, and

2. there are different degrees of similarity, namely copy, logical and structural equivalence between formulas.

The three degrees of equivalence are:

Copy Equivalence: Two formulas are considered copy equivalent, if they are the same, as if they resulted from a copy and paste operation[1]. Of course, retyping could deliver a similar result.

Logical Equivalence: Two formulas are logical equivalent, if they differ only in absolute cell references or constants. Logical equivalence is quite frequent for what-if analysis, and we found out that it often results from copy-paste and modify operations.

Structural Equivalence: Two formulas are structural equivalent, if they differ in absolute, relative cell references or constants. Hence, for two formulas to be structural equivalent, it is required that they apply the same functions in the same order to different arguments. For people used to procedural programming, structural equivalence might resemble to macros[2].

[1]In order to compare cell references in formulas we use the `R1C1` style to specify them. The number after the `R` denotes how many rows to move up or down from the referencing cell, whereas the `C` components denotes how many columns to go to the left or right. E.g. the relative reference `R-3C2` in the cell `D5` points to cell `F2`, in the cell `B4` it would reference `D1`. This notation was the one originally used in Visicalc and is still used as the internal representation in many modern spreadsheet systems.

[2]We use the term *macro* here not in the spreadsheet context, but in the context of procedural languages, e.g. a macro in `C`

	A	B	C	D	E	F
1						
2		Nr.	Balance	In	Out	New Balance
3		1	1000	50	70	=C3+D3-E3
4		=B3+1	=F3	20	50	=C4+D4-E4
5		=B4+1	=F4	40	90	=C5+D5-E5
6		=B5+1	=F5	140	30	=C6+D6-E6
7		=B6+1	=F6	10	50	=C7+D7-E7
8		=B7+1	=F7	70	10	=C8+D8-E8
9						

Figure 1: Cells in the same logical area are shaded equally, whereas semantic units are delimited by a thick border. All semantic units with more than one element are in the same semantic class, parametrisation for the semantic classes is $(d_h, d_v, d_{Man}) = (1, 0, 1)$

Further, all numeric constants, string constants and empty cells are assigned to a corresponding logical area.

In the rest of this paper we will not stress that the assignment of a cell to a logical area depends on its formula. If it is clear from the context we will state only that cell c_1 is copy equivalent with cell c_2, meaning that their formulas are copy equivalent.

A subset of the spreadsheet containing all cells with e.g. structural equivalent formulas, is called a structural equivalent logical area.

It is easy to see that there exists an order between the different equivalence criteria, formulas that are copy equivalent are also logical and structural equivalent. This property has shown very helpful for actual auditing of spreadsheets, as it enables the auditor to detect outliers quickly, e.g. a set of cells which are copy equivalent and a single cell that is only logical equivalent to all cells in the set might be a hint for a hot-spot.

The effectiveness of logical areas was empirically verified in a field audit [9]. It turned out that the concept is effective and the main strength is the ease of use and comprehensibility. However, there were limits in the scalability of the approach that often delivered complex abstractions for some sheets. We found mainly two reasons for the complexity of the abstractions. The first one was that for large spreadsheets logical areas sometimes delivered counter-intuitive groups, e.g. if the users create a spreadsheet by copying rows with different content, the logical areas will be formed by columns. An example of this is shown in Figure 1, where cells in the same logical area are shaded in the same colour. We argue, that it is very likely that a spreadsheet like this could be created by copying the row 4 down into all consecutive rows, rather than by copying cell by cell.

Another drawback was that logical areas did not take the spatial position of cells into account at all. Although an advantage in some case, e.g. it allows to spot regular patterns of similar cells, it can result in confusion for huge spreadsheets. Both these problems were alleviated by the concept of semantic classes that extends logical areas.

Another problem is the restricted scope of this technique. It is only useful for spreadsheets with many similar cells. However, in many applications formulas occur only once or twice, or not in a regular pattern at all. We figured out that we need an orthogonal approach for the latter case, which is presented in the Section 2.3.

2.2 Semantic Classes

A semantic class can be described as a re-occurring block where cells on the same relative position in the block are in the same logical area.

Blocks with similar cells on the same relative positions, consecutively called semantic units, have to satisfy certain geometric conditions that can restrict their horizontal and vertical extension as well as the size of gaps in these blocks. Originally, the geometrical conditions had to be supplied by the users by means of three parameters: d_h , d_v and d_{Man} . The first two specific the maximal size of gaps in the semantic unit, either horizontally (d_h) or vertically (d_v). Thus, by setting d_h to 1 and d_v to 0 users can require semantic units to consist of horizontally adjacent cells. Setting d_h to 2 and d_v to 0 allows semantic units to consist of horizontally adjacent cells, with gaps spanning at most one cell. In [11, 12] a more advanced approach is presented that uses layout information and labels to make guided guesses about the geometrical shape and extent of semantic units and, hence, the parametrisation is not necessary anymore.

In order to group a couple of semantic units into a semantic class, they are required to be similar. Two semantic units are considered similar, if they have an identical geometrical shape and extent, and all the cells on same relative positions in the semantic units are in the same logical area.

Semantic classes have the advantage that can deal very good with regular large spreadsheets, and small effects, e.g. a single deviating cell, will cause an effect on the final abstraction. In discussions with end users we figured out the further advantage, that this abstraction technique will result in abstract units that correspond to the way the users actually created the spreadsheet, i.e. copying rows will result in semantic units that consist of the actual row, not the other way round as it might happen with logical areas. In Figure 1 the semantic units are framed with a thick border.

The remaining disadvantage is the required parametrisation. Users have to have a basic understanding of the structure of the spreadsheet in advance, before they even apply this technique. We found out that users have to be taught in this concept before they can successfully use it. This was not the case with logical areas. However, a recent master thesis [11] overcame some of these limitations and automated the creation of semantic classes based on some heuristics.

2.2.1 Detecting Hot Spots

There are basically three known heuristics to detect hot spots of a spreadsheet program based on this abstraction[10]. The most straight-forward one is to look for a regular pattern of distribution of semantic units belonging to the same semantic class on the spreadsheet. Wherever that pattern, if any exists, is disrupted, a closer investigation is necessary.

A second strategy relies on the fact, that most errors are rather small deviations from a correct formula, e.g. a mis-reference or a wrong constant. Thus, if there is a group of semantic units that is, for instance copy-equivalent, but there are a few outliers, that are only logical-equivalent, it has to be investigated, whether an error is the source of this difference.

The third strategy relies on the inspection of the so-called SRG_{SC}. The SRG_{SC} is a directed graph of dependencies between semantic classes. Each node in the SRG_{SC} is a semantic class, and there is an edge from one node to another, if a cell, that belongs to a semantic unit that is in the first semantic class (represented by the target node) references a cell in a semantic unit of the semantic class that is represented by the source node. This graph reflects the cell dependencies of the original spreadsheet, but on a higher level of abstraction. In [10, 7] we suggest a fish-eye view approach to auditing based on the SRG_{SC}. Some of the new auditing heuristics presented in Section 3 make use of the SRG_{SC} as well, but, in contrast to the existing approaches, the ones suggested here can be carried out automatically.

2.3 Data Modules

Spreadsheet programs have some basic characteristics of data flow programs and of graph-reduction programs, too (see [13]). Thus, the data-dependency graph, subsequently called DDG, of a spreadsheet program has an important role for its execution. The DDG is a directed, acyclic graph, where every node represents a cell of the spreadsheet program. There is an edge between two nodes, if the cell represented by the target node is referencing the cell represented by the source node. Vertices that are not the source of any edge are called sink nodes.

To grasp the idea, one can assume that a data module is a set of cells with a distinguished result cell, that is transitively dependent on all cells in the data module. Cells that are outside the data module may only reference its result cell. Broadly speaking, a data module is a subgraph of the DDG, that has only a single sink node, namely its result cell. The result cell of such a data module is either a sink node of the DDG, i.e. a result cell of the spreadsheet program, or a node that is connected to more than one data module. For a formal definition and algorithms to recover data modules from existing spreadsheets, see [6]

Spreadsheet programmers are not forced to follow a certain design paradigm and identify data-modules, but we try to identify data modules by analysing the finished spreadsheet. Cells that are not part of a specific data module may reference only its result cell. Obviously, this definition is recursive, but because of the hierarchical organisation of a DDG and its finiteness, this is not a problem. As the data modules are not a-priori known, we have developed a way to recover them out of the spreadsheet's DDG. The recovery of data modules will start assuming the spreadsheet's result cells to be data modules and adds all cells that are only referenced by one data module to the referencing data module. A cell that transitively contributes to more than one data module is assumed to be the starting point of a new data module and will be treated in the same way.

Before the DDG can be partitioned into such data modules, the result cells have to be identified. Obviously, not all sink nodes of the DDG have the semantics of a result of the spreadsheet program, e.g. check-sums. In contrast to conventional programming where intermediate results are not displayed and each subroutine has a well defined result, in a spreadsheet each intermediate result is visible to the user and to all the other formulas. Sometimes, calculations are deliberately formulated in a more complicated way in order to obtain some desired intermediate results.

Cells of a spreadsheet program are either auxiliary, intermediate or result cells. For sure, cells that are not further referenced by other cells can be considered result cells, because we know that users place them on the spreadsheet, because they want to see their contents. If they would not like to see the displayed value, they had not introduced this cell. Therefore, it seems legitimate to consider DDG sink nodes as result cells, and start constructing data modules by searching cells, that influence a specific result. As a matter of fact, it is often the case, that sink nodes in the DDG are not the real results, but check-sums. In this case, the check-sums have to be removed manually, and the remaining DDG is then analysed.

2.3.1 Detecting Hot Spots

Data modules are particularly useful to identify errors due to mis-references. If a planned cell reference is not part of a formula,

	A	B	C	D	E	F
1						
2	Costs per Unit:		1			
3	Revenue per Unit:		3.5			
4						
5		1st Quarter	2nd Quarter	3rd Quarter	4th Quarter	**Year**
6						
7	Units Sold	12000	9000	10000	17231	**48231**
8	Total Costs	12000	9000	10000	17231	**48231**
9	Total Revenue	42000	31500	35000	60308.5	**168809**
10	Result	30000	22500	25000	43077.5	**120578**

Figure 2: Example Spreadsheet, displaying results.

	A	B	C	D	E	F
1						
2	Costs per Unit:		1			
3	Revenue per Un		3.5			
4						
5		1st Quarter	2nd Quarter	3rd Quarter	4th Quarter	**Year**
6						
7	Units Sold	12000	9000	10000	17231	**=SUM(B7:E7)**
8	Total Costs	=$C$2*B7	=$C$2*C7	=$C$2*D7	=$C$2*E7	**=SUM(B8:E8)**
9	Total Revenue	=$C$3*B7	=$C$3*C7	=$C$3*D7	=$C$3*E7	**=SUM(B9:E9)**
10	Result	=B9-B8	=C9-C8	=D9-D8	=E9-E8	**=SUM(B10:E10)**

Figure 3: Example Spreadsheet, displaying formulas.

the data module will split up into two different modules. The opposite case, that of a cell reference that should not be part of a formula, might lead to the merge of two unrelated data modules. Hence, auditors have to watch out for superfluous and absent data modules.

Subsequently, the cell where the result of the superfluous data module should have been referenced has to be identified and corrected. The opposite case is more difficult. If an expected data module is not part of the visualisation, auditors have to look for the cell where the missing data module is erroneously referenced. Although fault tracing is more troublesome, the presence of an error can be easily detected.

In contrast, certain kinds of errors that are easily discovered by other techniques do not influence the resulting data modules at all. E.g., wrong operators or mis-references to cells in the same data module, will influence the result of a data module, but not the assignment of cells to a data module, as only the data dependencies are taken into account.

A different auditing strategy makes again use of the fact that we can generate a compressed but semantically equivalent representation of the DDG. In the so generated SRG_{DM}, each data module is a node and there is an edge between data modules, if one references the result cell of the other one. Assuming that the original DDG is acyclic, the SRG_{DM} will be acyclic, too. The SRG_{DM} can be used to generate a fish-eye view of the spreadsheets, if we replace one of the data modules by the subgraph of the DDG that it corresponds to. Thus, we can have a very detailed look at a certain part of the spreadsheet, without being bothered by unnecessary details, but still having an eye on the context of the part we are currently examining.

3. FOUR NEW HEURISTICS

In this section we will introduce four new heuristics that are meant to help the auditor by automatically detecting hot-spots by means of inspecting the SRG_{SC} or SRG_{DM}. The heuristics are well suitable for the examination of large spreadsheet programs because they can be easily automated. However, as this is work in progress it is not part of the toolkit developed so far [8, 12].

3.1 Heuristics 1 and 2: Aggregation Examination

Usually spreadsheet programs contain different parts yielding intermediate results, that are then aggregated, either directly by means of a an aggregation function, like `SUM`, `MIN` or `AVG`, or indirectly, e.g. by sequentially applying the same operator, into a single result. The heuristics introduced subsequently exploit the concept of *aggregation equivalence* that is introduced in [7], and also used in many visualisation tools. Below, we introduce two heuristics that exploit this common pattern.

3.1.1 Heuristic 1: Semantic Class Aggregation

Assume that a cell references a set of cells that belongs to different semantic units in the same semantic class. In Figure 2 the year's result is computed by summing up the results of each quarter. In our example, each quarter has been identified as a semantic unit of the same semantic class by the automatic algorithm suggested in [11]. The formula view of the same spreadsheet, shown in Figure 3, reveals that the formulas used in each row are the same. As each semantic unit has similar, i.e. in our case copy-equivalent, formulas on the same relative positions they are considered similar as well, and thus form a semantic class.

Heuristic 1 assumes a hot-spot, if the majority of arguments to an aggregation function are cells in different but similar semantic units, but there is at least one cell referenced that is not in a similar semantic unit. This goes beyond the capabilities that are currently available e.g. in Excel, as this heuristic does not require equal formulas in the aggregated cells and does not require them to be in a compact spatial area.

We assume that there is a certain user-specified threshold to determine what should be considered a majority. For instance, in Figure 4, a threshold of 75% would identify hot-spots in cells `F7`, `F8`, `F9` and `F10`. This seems strange at first sight, because only `F7` references three similar and one different cells. However, each of the `SUM` formulas aggregates three similar semantic units and one outlier. Detecting the actual irregularity can be easily achieved by inspecting the set of cells not in the semantic class.

Obviously, this heuristic can be enforced by requiring that the arguments to the aggregation function should be in the same relative position in the referenced, similar semantic units. This would not make a difference in our case, but can be easily applied to detect mis-reference, that would go undetected by the un-enforced heuristic.

3.1.2 Heuristic 2: Data Module Aggregation

This heuristic is based on the often observed pattern in spreadsheets that the results of different data-modules is processed by aggregation formulas. It is very similar to Heuristic 1, but this time not based on semantic units, but on data modules. We suggest, that if a majority of the cells referenced by an aggregation formula are result cells of data modules, those cells that are not, are hot-spots and should be scrutinised.

In terms of the example presented above, each quarter would be a data module, if only a yearly result is calculated, and the other yearly figures are neglected as check-sums. In Figure 3, `B10`, `C10`, `D10` and `E10` would then be the result cells of data modules that are summed in `F10`[3]. However, if there would be any mis-reference, e.g. to `C9` instead of `C10`, or the figures of the second quarter where

[3]Assuming that the quarterly results are also referenced by at least one other formula.

	A	B	C	D	E	F
1						
2	Costs per Unit:		1			
3	Revenue per Unit		3.5			
4						
5		1st Quarter	2nd Quarter	3rd Quarter	4th Quarter	**Year**
6						
7	Units Sold	12000	9000	=C7*1.1	17231	**=SUM(B7:E7)**
8	Total Costs	=$C$2*B7	=$C$2*C7	=$C$2*D7	=$C$2*E7	**=SUM(B8:E8)**
9	Total Revenue	=$C$3*B7	=$C$3*C7	=$C$3*D7	=$C$3*E7	**=SUM(B9:E9)**
10	Result	=B9-B8	=C9-C8	=D9-D8	=E9-E8	**=SUM(B10:E10)**

Figure 4: Example Spreadsheet with hot-spots due to an irregularity in D7.

misaligned, an irregularity in F10 is detected.

Heuristic 2 and Heuristic 1 can be synthesised to form an even stronger indicator for a hot-spot: If the majority of cells that are aggregated by an aggregation formula are the result cells of data modules, that are also similar semantic units, outliers very strongly indicate a hot-spot.

3.2 Heuristics 3 and 4: SRG_{SC} Links

The heuristics 3 and 4, which are subsequently presented, examine links between semantic units in different semantic classes. If a regular pattern can be identified, i.e. the majority of semantic units in one class references semantic units in a specific semantic class, outliers might indicate hot-spots.

3.2.1 Heuristic 3: Weak SRG_{SC} Link

There is a weak SRG_{SC} link between two semantic classes, SC_1 and SC_2, if a majority of semantic units in SC_1 references cells in semantic units in SC_2. To determine, what is considered a majority, parametrisation by the user is required. Those semantic units in SC_1 that do not reference semantic units in SC_2 are considered hot-spots and should be examined.

We want to underline that we also assume a link between two semantic classes SC_1 and SC_2, if the majority of semantic units in SC_1 references the same semantic unit of SC_2. In Figure 5 we show three semantic classes in different shades of gray, the semantic units are framed by a bold border. Cells with white background and no border are singular semantic units that form singular semantic classes. There is a weak SRG_{SC} link between the light gray shaded semantic class and both, the dark gray shaded class and the singular class formed by cell B2, but for different reasons.

The link to the dark gray shaded semantic class is due to references to different semantic units, whereas B2 obviously relates because of references to a single member. If we assume a threshold of more than 33%, there will be no link between the light gray semantic class and the singular semantic class formed by B1, because it is referenced only by one out of three semantic units in the class.

As semantic classes are an abstraction mechanism for logical areas, the heuristic can also be applied to them, introducing a weak SRG_{LA}[4] link between to logical areas l_1 and l_2, if the majority of cells in l_1 references cells in l_2.

3.2.2 Heuristic 4: Strong SRG_{SC} Links

There is a strong SRG_{SC} link between two semantic classes SC_1 and SC_2, if a majority of semantic units in SC_1 references cells **on the same relative positions** in semantic units in SC_2 and

[4]The SRG_{LA} is a directed graph where each logical area in the spreadsheet is represented by a node and there is an edge from node n_1 to node n_2, if a cell in n_2 references a cell in n_1.

	A	B	C	D
1	Income 2004	1000	Credit/Child	75
2	Inflation	0.02	Children	2
3				
4	Forecast			
5		Income	Taxes	
6	2005	=B1*(1+B$2)	=B6*0.2-C12	
7	2006	=B6*(1+B$2)	=B7*0.2-C13	
8	2007	=B7*(1+B$2)	=B8*0.2-C14	
9				
10				
11		Credit/Child	Credit	
12	2005	=D1*(1+B$2)	=B12*D$2	
13	2006	=B12*(1+B$2)	=B13*D$2	
14	2007	=B13*(1+B$2)	=B14*D$2	

Figure 5: Simple tax forecast

SC_2 does not consist of singular semantic units, and contains more than one semantic unit. Again, parametrisation is required to determine what a majority is.

In Figure 5 the link between the light gray shaded and the dark gray shaded semantic classes is a strong link, as all semantic units in the first reference cells on the same relative position in semantic units of the latter. Semantic units in SC_1 that do not reference semantic units in SC_2 at all or reference to cells on different relative positions in semantic units of SC_2 are considered hot-spots.

We consider irregularities detected by heuristic 4 to be more significant than those detected by heuristic 3, as heuristic 4 is obviously a more restricted form of the former.

4. DISCUSSION

There are quite a few things that are outside the domain of our work and hence are not touched in this paper. The first, and most obvious is that we are discussing only correctness of formulas- we cannot make any statement about the correctness of the values that are used as input to these formulas. Therefore, we must refer to other techniques.

The approach we propagate aims to support users and auditors by highlighting irregularities in the spreadsheet. These irregularities can be introduced on purpose and not indicate any error at all, whereas other errors might be propagated by subsequent copy-and-paste operations and form a regular pattern by themselves. Hence, in order to make an absolute statement about the correctness of a given spreadsheet program, we have to refer to approaches that promote exhaustive testing of the spreadsheet, like [1, 21] or cell-by-cell auditing, like [19]. Visualisation approaches will generally only help the auditors by providing a better understanding of the underlying spreadsheet and highlighting some irregularities.

However, in practice there is only limited time and resources available for the checking of a particular spreadsheet. Applying systematic testing approaches to the areas that are identified as hot-spots might be more promising than to apply them to an arbitrary part of the spreadsheet- assuming that we do not have enough resources to exhaustively test the whole program.

The heuristics presented in this paper are meant to offer quick automated checks for a spreadsheet. As there already exists a toolkit to extract logical areas, semantic classes, data modules and the associated SRGs from spreadsheet programs, we aim to extend the toolkit with these automatic checks. Hence, even large spreadsheets can be quickly checked and the attention of the auditor- or

the programmer- can be directed to the hot-spots. We also suggest to offer an assistant to spreadsheet users that alerts them, whenever a change to a formula might lead to a hot-spot by any of these heuristics.

Next steps in our research will be the implementation of these heuristics in our prototype and, subsequently, gathering of experimental data. As there is still a threshold parameter required for any of the suggested heuristics, it will also be worth investigating the influence of different values on the rate of identified errors and hot-spots that are not errors at all.

5. REFERENCES

[1] Y. Ayalew. *Spreadsheet Testing Using Interval Analysis*. PhD thesis, Universität Klagenfurt, Universitätsstrasse 65–67, A-9020 Klagenfurt, Austria, November 2001.

[2] Y. Ayalew, M. Clermont, and R. Mittermeir. Detecting errors in spreadsheets. In *Spreadsheet Risks, Audit and Development Methods*, volume 1, pages 51–62, AAAAAA, 7 2000. EuSpRIG, University of Greenwich.

[3] R. Butler. Is This Spreadsheet a Tax Evader ? How H. M. Customs & Excise Test Spreadsheet Applications. In *Proceedings of the 33rd Hawaii International Conference on System Sciences - 2000*, volume 33, 2000.

[4] D. Chadwick, K. Rajalingham, B. Knight, and D. Edwards. An Approach to the Teaching of Spreadsheets Using Software Engineering Concepts. In *Proceedings of the 4th International Conference on Software Process Improvement, Research, Education and Training INSPIRE'99*, pages 261–273, 1999.

[5] Y. E. Chan and V. C. Storey. The use of spreadsheets in organizations: Determinants and consequences. *Information & Management*, 31:119–134, 1996.

[6] M. Clermont. Analyzing large spreadsheet programs. In *Proceedings of the10th Working Conference on Reverse Engineering*. IEEE, 2003.

[7] M. Clermont. *A Scalable Approach to Spreadsheet Visualization*. PhD thesis, Universität Klagenfurt, Universitätsstrasse 65–67, A-9020 Klagenfurt, Austria, 2003.

[8] M. Clermont. A toolkit for scalable spreadsheet visualisation. In *Risk Reduction in End User Computing*, volume 4. EuSpRIG, 7 2004.

[9] M. Clermont, C. Hanin, and R. Mittermeir. A Spreadsheet Auditing Tool Evaluated in an Industrial Context . In *Spreadsheet Risks, Audit and Development Methods*, volume 3, pages 35–46. EUSPRIG, 7 2002.

[10] M. Clermont and R. Mittermeir. Auditing large spreadsheet programs. In *ISIM'03, Proceedings of the 6th International Conference*, pages 87–97, 2003.

[11] S. Hipfl. Spreadsheet-Visualisierung unter Berueecksichtigung von Layout-Information. Master's thesis, University Klagenfurt, 1999.

[12] S. Hipfl. Using layout information for spreadsheet visualization. In *Risk Reduction in End User Computing*, volume 4. EuSpRIG, 7 2004.

[13] K. Hodnigg, M. Clermont, and R. Mittermeir. Computational models of spreadsheet development: Basis for educational approaches. In *Risk Reduction in End User Computing*, volume 4. EuSpRIG, 7 2004.

[14] T. Isakowitz, S. Shocken, and H. C. Lucas. Toward a Logical/Physical Theory of Spreadsheet Modeling. *ACM Transactions on Information Systems*, 13(1):1–37, 1995.

[15] B. Knight, D. Chadwick, and K. Rajalingham. A structured methodology for spreadsheet modelling. In *Spreadsheet Risks, Audit and Development Methods*, volume 1, pages 43–50. EuSpRIG, University of Greenwich, 7 2000.

[16] R. Mittermeir and M. Clermont. Finding High-Level Structures in Spreadsheets. In *Proceedings of the 9th Working Conference on Reverse Engineering*, 2002.

[17] P. O'Beirne. Spreadsheet errors, news stories about spreadsheets with costly mistakes. http://www.eusprig.org/stories.html, 2005.

[18] G. J. O'Brien and W. D. Wilde. Australian managers' perceptions, attitudes and use of information technology. *Information and Software Technology*, 38:783–789, 1996.

[19] R. Panko and R. P. Halverson. Are Two Heads Better than One? (At Reducing Errors in Spreadsheet Modeling). *Office Systems Research Journal*, 1997.

[20] R. R. Panko. What we know about spreadsheet errors. *Journal of End User Computing: Special issue on Scaling Up End User Development*, 10(2):15–21, Spring 1998.

[21] J. Reichwein, G. Rothermel, and M. Burnett. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. In *Proceedings of the 2nd Conference on domain-specific languages*, volume 2, pages 25–38. ACM, 2000.

[22] B. Ronen, M. Palley, and H. Lucas. Spreadsheet analysis and design. *Communication of the ACM*, 32(1):84–93, January 1989.

[23] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, and G. Rothermel. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In *ICSE 2000 Proceedings*, pages 230–239. ACM, 2000.

[24] J. Sajaniemi. Modeling spreadsheet audit: A rigorous approach to automatic visualization. *Journal of Visual Languages and Computing*, 11(1):49–82, 2000.