On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets

Birgit Hofer[1], André Riboira[2], Franz Wotawa[1], Rui Abreu[2], and Elisabeth Getzner[1]

[1] Institute for Software Technology, Graz University of Technology, Graz, Austria
{bhofer,wotawa}@ist.tugraz.at, elisabeth.getzner@student.tugraz.at
[2] Department of Informatics Engineering, Faculty of Engineering of University of Porto, Porto, Portugal
andre.riboira@fe.up.pt, rui@computer.org

Abstract. Spreadsheets are by far the most prominent example of end-user programs of ample size and substantial structural complexity. In addition, spreadsheets are usually not tested very rigorously and thus comprise faults. Locating faults is a hard task due to the size and the structure, which is usually not directly visible to the user, i.e., the functions are hidden behind the cells and only the computed values are presented. Hence, there is a strong need for debugging support. In this paper, we adapt three program-debugging approaches that have been designed for more traditional procedural or object-oriented programming languages. These techniques are Spectrum-based Fault Localization, Spectrum-Enhanced Dynamic Slicing, and Constraint-based Debugging. Beside the theoretical foundations, we present a more sophisticated empirical evaluation including a comparison of these approaches. The empirical evaluation shows that Sfl (Spectrum-based Fault Localization) and Sendys (Spectrum ENhanced Dynamic Slicing) are the most promising techniques.

Keywords: End-User debugging, spreadsheets, spectrum-based fault localization, model-based debugging.

1 Introduction

Spreadsheet tools, such as Microsoft Excel, iWork's Numbers, and OpenOffice's Calc, can be viewed as programming environments for non-professional programmers [1]. In fact, these so-called "end-user" programmers vastly outnumber professional ones: the US Bureau of Labor and Statistics estimates that more than 55 million people use spreadsheets and databases at work on a daily basis [1]. Despite this trend, as a programming language, spreadsheets lack support for abstraction, testing, encapsulation, or structured programming. As a consequence, spreadsheets are error-prone. Numerous studies have shown that existing spreadsheets contain redundancy and errors at an alarmingly high rate [2].

Furthermore, spreadsheets are applications created by single end-users without planning ahead of time for maintainability or scalability. Still, after their initial creation, many spreadsheets turn out to be used for storing and processing increasing amounts of data as well as supporting increasing numbers of users over long periods of time. Therefore, debugging (i.e., locating the cell(s) that are responsible for the wrong output in a given cell) can be a rather cumbersome task, requiring substantial time and effort.

In spite of having the potential to benefit from recent developments in the software engineering domain, the truth is that only a few attempts have been made to adapt software engineering techniques to the spreadsheet world. The objective of this paper is to advance the state of the art in spreadsheet debugging by applying popular, mature techniques developed to analyze software systems. Such techniques will have a positive impact in the overall quality of spreadsheets.

In this paper, we adapt three program-debugging approaches that have been designed for more traditional procedural or object-oriented programming languages. In particular, we describe how to modify traditional fault localization techniques in order to render them applicable to the spreadsheet world. We consider the following techniques in our study: Spectrum-based Fault Localization (Sfl) [3], Spectrum-enhanced dynamic slicing (Sendys) [4], and Constraint-based Debugging (ConBug) [5]. We evaluate the efficiency of the approaches using real spreadsheets taken from the EUSES Spreadsheet Corpus [6].

The remainder of the paper is organized as follows: Section 2 deals with the related work. In addition, existing spreadsheet debugging and testing techniques are discussed. Section 3 deals with the syntax and semantics of spreadsheets. Furthermore, the Spreadsheet Debugging problem is defined. Section 4 explains the changes that have to be made in order to use the existing debugging techniques for debugging of spreadsheets. Three traditional debugging techniques are explained in detail. Section 5 deals with the setup and the results of the empirical evaluation. Finally, Section 6 concludes this paper and presents ideas for future empirical evaluations.

2 Related Work

Since spreadsheet *developers* are typically end-users without significant background in computer science, there has been considerable effort to adapt software engineering principles to form a spreadsheet engineering discipline (e.g., [7–11]).

Some of the work presented in this paper is based on model-based diagnosis [12], namely its application to (semi-)automatic debugging (e.g., [13]). In contrast to previous work, the work presented in this paper does not use logic-based models of programs but instead uses a generic model which can be automatically computed from the spreadsheet. A similar approach has been presented recently to aid debuggers in pinpointing software failures [14]. Moreover, Jannach and Engler presented a model-based approach [15] to calculate possible error causes in spreadsheets. This approach uses an extended hitting-set algorithm and user-specified or historical test cases and assertions.

GoalDebug [16, 17] is a spreadsheet debugger for end users. Whenever the computed output of a cell is incorrect, the user can supply an expected value for a cell, which is employed by the system to generate a list of change suggestions for formulas that, when applied, would result in the user-specified output. In [16] a thorough evaluation of the tool is given. GoalDebug employs an approach similar to the constraint-based approach presented in this paper.

Spreadsheet testing is closely related to debugging. In the WYSIWYT system users can indicate incorrect output values by placing a faulty token in the cell. Similarly, they can indicate that the value in a cell is correct by placing a correct token [18]. When a user indicates one or more program failures during this testing process, fault localization techniques direct the user's attention to the possible faulty cells. However, WYSIWYT does not provide any suggestions for how to change erroneous formulas.

3 Basic Definitions

A spreadsheet is a matrix comprising cells. Each cell is unique and can be accessed using its corresponding column and row number. For simplicity, we assume a function φ that maps the cell names from a set $CELLS$ to their corresponding position (x, y) in the matrix where x represents the column and y the row number. The functions φ_x and φ_y return the column and row number of a cell respectively.

Aside from a position, each cell $c \in CELLS$ has a value $\nu(c)$ and an expression $\ell(c)$. The value of a cell can be either undefined ϵ, an error $\perp$, or any number, boolean or string value. The expression of a cell $\ell(c)$ can either be empty or an expression written in the language $\mathcal{L}$. The value of a cell c is determined by its expression. If no expression is explicitly declared for a cell, the function ℓ returns the value ϵ.

Areas are another important basic element of spreadsheets. An area is a set consisting of all cells that are within the area that is spanned by the cells $c_1, c_2 \in CELLS$. Formally, we define an area as follows:

$$c_1{:}c_2 \equiv_{def} \left\{ c \in CELLS \;\middle|\; \begin{array}{l} \varphi_x(c_1) \leq \varphi_x(c) \leq \varphi_x(c_2) \;\; \& \\ \varphi_y(c_1) \leq \varphi_y(c) \leq \varphi_y(c_2) \end{array} \right\}$$

Obviously, every area is a subset of the set of cells ($c_1{:}c_2 \subseteq CELLS$). After defining the basic elements of spreadsheets, we introduce the language $\mathcal{L}$ for representing expressions that are used to compute values for cells. For reasons of simplicity, we do not introduce all available functions in today's spreadsheet implementations. Instead and without restricting generality, we make use of simple operators on cells and areas. Extending the used operators with new ones is straightforward.

The introduced language takes the values of cells and constants together with operators and conditionals to compute values for other cells. The language is a functional language, i.e., only one value is computed for a specific cell. Moreover, we do not allow recursive functions. First, we define the syntax of $\mathcal{L}$.

Definition 1 (Syntax of $\mathcal{L}$). *We define the syntax of $\mathcal{L}$ recursively as follows:*

- *Constants k representing ϵ, number, boolean, or string values are elements of $\mathcal{L}$ (i.e., $k \in \mathcal{L}$).*
- *All cell names are elements of $\mathcal{L}$ (i.e., $CELLS \subset \mathcal{L}$).*
- *If e_1, e_2, e_3 are elements of the language ($e_1, e_2, e_3 \in \mathcal{L}$), then the following expressions are also elements of $\mathcal{L}$:*
  - *(e_1) is an element of $\mathcal{L}$.*
  - *If o is an operator ($o \in \{+, -, *, /, <, =, >\}$), then $e_1\ o\ e_2$ is an element of $\mathcal{L}$.*
  - ***if(**e_1**;** e_2**;** e_3**)** is an element of $\mathcal{L}$.*
- *If c_1**:**c_2 is an area, then **sum(**c_1**:**c_2**)** is an element of $\mathcal{L}$.*

Second, we define the semantics of $\mathcal{L}$ by introducing an interpretation function $[\![\cdot]\!]$ that maps an expression $e \in \mathcal{L}$ to a value. The value is ϵ if no value can be determined or $\perp$ if a type error occurs. Otherwise it is either a number, a boolean, or a string.

Definition 2 (Semantics of $\mathcal{L}$). *Let e be an expression from $\mathcal{L}$ and ν a function mapping cell names to values. We define the semantic of $\mathcal{L}$ recursively as follows:*

- *If e is a constant k, then the constant is given back as result, i.e., $[\![e]\!] = k$.*
- *If e denotes a cell name c, then its value is returned, i.e., $[\![e]\!] = \nu(c)$.*
- *If e is of the form (e_1), then $[\![e]\!] = [\![e_1]\!]$.*
- *If e is of the form $e_1\ o\ e_2$, then its execution is defined as follows:*
  - *If either $[\![e_1]\!] = \perp$ or $[\![e_2]\!] = \perp$, then $[\![e_1\ o\ e_2]\!] = \perp$.*
  - *else if either $[\![e_1]\!] = \epsilon$ or $[\![e_2]\!] = \epsilon$, then $[\![e_1\ o\ e_2]\!] = \epsilon$.*
  - *else if $o \in \{+, -, *, /, <, =, >\}$, then*

$$[\![e_1\ o\ e_2]\!] = \begin{cases} [\![e_1]\!]\ o\ [\![e_2]\!] & \text{if all sub-expressions evaluate to a number} \\ \perp & \text{otherwise} \end{cases}$$

- *If e is of the form **if(**e_1**;** e_2**;** e_3**)**, then*

$$[\![e]\!] = \begin{cases} [\![e_2]\!] & \text{if } [\![e_1]\!] = \textbf{true} \\ [\![e_3]\!] & \text{if } [\![e_1]\!] = \textbf{false} \\ \epsilon & \text{if } [\![e_1]\!] = \epsilon \\ \perp & \text{otherwise} \end{cases}$$

- *If e is of the form **sum(**c_1**:**c_2**)**, then*

$$[\![e]\!] = \begin{cases} \sum_{c \in c_1:c_2} [\![c]\!] & \text{if all cells in } c_1\text{:}c_2 \text{ have a number or } \epsilon \text{ (treated as 0) as value} \\ \perp & \text{otherwise} \end{cases}$$

Frequently, we require information about cells that are used as input in an expression. We call such cells *referenced cells*.

Definition 3 (Referenced cell). *A cell c is said to be referenced by an expression $e \in \mathcal{L}$, if and only if c is used in e.*

We furthermore introduce a function $\rho : \mathcal{L} \mapsto 2^{CELLS}$ that returns the set of referenced cells. Formally, we define ρ as follows:

Definition 4 (The function ρ). *Let $e \in \mathcal{L}$ be an expression. We define the referenced cells function ρ recursively as follows:*

- *If e is a constant, then $\rho(e) = \emptyset$.*
- *If e is a cell c, then $\rho(e) = \{c\}$.*
- *If $e =$ **(**e_1**)**, then $\rho(e) = \rho(e_1)$.*
- *If $e = e_1\ o\ e_2$, then $\rho(e) = \rho(e_1) \cup \rho(e_2)$.*
- *If $e =$ **if(**e_1**;** e_2**;** e_3**)**, then $\rho(e) = \rho(e_1) \cup \rho(e_2) \cup \rho(e_3)$.*
- *If $e =$ **sum(**c_1**:**c_2**)**, then $\rho(e) = c_1$**:**c_2.*

A spreadsheet is a matrix of cells comprising values and expressions written in a language $\mathcal{L}$. In addition, we know that the values of cells are determined by their expressions. Hence, we can state that $\forall c \in CELLS : \nu(c) = [\![\ell(c)]\!]$ must hold. Unfortunately, we face two challenges: (1) In all of the previous definitions, the set of cells need not be of finite size. (2) There might be a loop in the computation of values, e.g. a cell c with $\ell(c) = c{+}1$. In this case, we are not able to determine a value for cell c. In order to solve the first challenge, we formally restrict spreadsheets to comprise only a finite number of cells.

Definition 5 (Spreadsheet). *A countable set of cells $\Pi \subseteq CELLS$ is a spreadsheet if all cells in Π have a non empty corresponding expression or are referenced in an expression, i.e., $\forall c \in \Pi : (\ell(c) \neq \epsilon) \vee (\exists c' \in \Pi : c \in \rho(\ell(c')))$.*

In order to solve the second challenge, we have to limit spreadsheets to loop-free spreadsheets. For this purpose, we first introduce the notation of data dependence between cells, and furthermore the data dependence graph, which represents all dependencies occurring in a spreadsheet.

Definition 6 (Direct dependence). *Let c_1, c_2 be cells of a spreadsheet Π. The cell c_2 depends directly on cell c_1 if and only if c_1 is used in c_2's corresponding expression, i.e., $dd(c_1, c_2) \leftrightarrow (c_1 \in \rho(\ell(c_2)))$.*

The direct dependence definition states the data dependence between two cells. This definition can be extended to the general case in order to specify indirect dependence. In addition, this dependence definition immediately leads to the definition of a graph that can be extracted from a spreadsheet.

Definition 7 (Data dependence graph (DDG)). *Let Π be a spreadsheet. The data dependence graph (DDG) of Π is a tuple (V, A) with:*

- *V as a set of vertices comprising exactly one vertex n_c for each cell $c \in \Pi$*
- *A as a set comprising arcs (n_{c_1}, n_{c_2}) if and only if there is a direct dependence between the corresponding cells c_1 and c_2 respectively, i.e. $A = \bigcup (n_{c_1}, n_{c_2})$ where $n_{c_1}, n_{c_2} \in V \wedge dd(c_1, c_2)$.*

From this definition, we are able to define general dependence between cells. Two cells of a spreadsheet are dependent if and only if there exists a path between the corresponding vertices in the DDG. In addition, we are able to further restrict spreadsheets to face the second challenge.

Definition 8 (Feasible spreadsheet). *A spreadsheet Π is feasible if and only if its DDG is acyclic.*

From here on, we assume that all spreadsheets of interest are feasible. Hence, we use the terms spreadsheet and feasible spreadsheet synonymously. Standard spreadsheet programs like Excel rely on loop-free computations.

In this paper, we focus on testing and debugging of spreadsheets. In ordinary sequential programs, a test case comprises input values and expected output values. If we want to rely on similar definitions, we have to clarify the terms input, output and test case. Defining the input and output of feasible spreadsheets is straightforward by means of the DDG.

Definition 9 (Input, output). *Given a feasible spreadsheet Π and its DDG (V, A), then the input cells of Π (or short: inputs) comprise all cells that have no incoming edges in the corresponding vertex of Π's DDG. The output cells of Π (or short: outputs) comprise all cells where the corresponding vertex of the DDG has no outgoing vertex.*

$$\begin{aligned} inputs(\Pi) &= \{c | \nexists (n_{c'}, n_c) \in A\} \\ outputs(\Pi) &= \{c | \nexists (n_c, n_{c'}) \in A\} \end{aligned}$$

All cells of a spreadsheet that serve neither as input nor as output are called intermediate cells. With this definition of input and output cells we are able to define a test case for a spreadsheet and its evaluation.

Definition 10 (Test case). *Given a spreadsheet Π, then a tuple (I, O) is a test case for Π if and only if:*

- *I is a set of tuples (c, e) specifying input cells and their values. For each $c \in inputs(\Pi)$ there must be a tuple (c, e) in I where $e \in \mathcal{L}$ is a constant.*
- *O is a set of tuples (c, e) specifying expected output values. The expected output values must be constants of $\mathcal{L}$.*

In our setting, test case evaluation works as follows: First, the functions $\ell(c)$ of the input cells are set to the constant values specified in the test case. Subsequently, the spreadsheet is evaluated. Afterwards, the computed output values are compared with the expected values stated in the test case. If at least one computed output value is not equivalent to the expected value, the spreadsheet fails the test case. Otherwise, the spreadsheet passes the test case.

In traditional programming languages, test cases are separated from the source code. Usually, there are several test cases for one function under test. Each of the test cases calls the function with different parameters and checks the correctness of the returned values. However, test cases are only implicitly encoded into spreadsheets. This means, that test cases are not explicitly separated from the formulas under test. If the user wants to add an additional test case, he or she has to duplicate the spreadsheet. A duplication of a spreadsheet for testing purposes is unpractical since the duplicates have to be updated when the spreadsheet is modified or extended. Therefore, usually only one failing test case exists. Hence, we reduce the debugging problem for spreadsheets to handle only one test case.

Definition 11 (Spreadsheet debugging problem). *Given a spreadsheet Π and a failing test case (I, O), then the debugging problem is to find a root cause for the mismatch between the expected output values and the computed ones.*

We define the spreadsheet debugging problem as a fault localization problem. This definition implies that the following debugging approaches pinpoint certain cells of a spreadsheet as possible root causes of faults. However, the approaches do not make any suggestions how to change these parts. Alternatively, the debugging problem can be defined as a fault correction problem.

4 Debugging Approaches

Traditional procedural and object-oriented program-debugging techniques cannot be directly transferred to spreadsheets for the following reasons: In a spreadsheet paradigm, the concept of code coverage does not exist since there are no explicit lines of code like in traditional programming paradigms. Moreover, there is no concept of test execution.

Therefore, in order to use traditional program-debugging techniques on spreadsheets, we have to perform some modifications: the lines of code in a traditional programming paradigm are mapped to the cells of a spreadsheet. There are cells designed to receive user input, cells to process data (using spreadsheet formulas), and cells intended to display the results. As an alternative to the code coverage of traditional programming paradigms, we compute so-called *cones* (data dependencies of each cell).

Definition 12 (The function CONE). *Given a spreadsheet Π and a cell $c \in \Pi$, then we define the function* CONE *recursively as follows:*

$$\text{CONE}(c) = c \cup \bigcup_{c' \in \rho(c)} \text{CONE}(c')$$

The correctness of the output cells is determined either by the user, by comparing the results of the current spreadsheet Π with another spreadsheet considered correct, or by applying techniques to automatically detect "bad smells" [19].

With these modifications, we are able to apply three traditional fault localization techniques on spreadsheets. In the following subsections, we explain these debugging techniques.

4.1 Spreadsheets Spectrum-Based Fault Localization

In traditional programming paradigms, Spectrum-based fault localization (SFL) [3] uses code coverage data and the pass/fail result of each test execution of a given system under test (SUT) as input. The code coverage data [20] is collected from test cases by means of an instrumentation approach. This data is collected at runtime and is used to build a so-called hit-spectra matrix. A hit-spectra matrix A is a binary matrix where each column i represents a system component and each row j represents a test case. The content of the matrix a_{ij} represents whether component i was used (true) or not (false) during test execution j. The results of the test executions (pass/fail) are stored in an error vector E. The error vector is a binary array where each position i represents a test execution. The value of the error vector e at position i is true if the test case i failed, otherwise false.

SFL uses similarity coefficients to estimate the likelihood of a given software component being faulty. Similarity coefficients compute the relationship between each column of the matrix (representing a system component) and the error vector. This similarity coefficient and the failure probability of the corresponding system component are directly related [21]. The coefficients are used to create rankings of system components [22] or to create interactive visualizations of the SUT, revealing the most suspicious parts of the application's source code [23].

In the spreadsheet paradigm, we cannot use the coverage data of test executions. Instead, we use the cones of the output cells (see Definition 12). From the cones, the hit-spectra matrix can be generated (each row of the matrix has the dependencies of one output cell). The error vector represents the correctness of the output cells. The hit-spectra matrix and the error vector allow the use of any SFL algorithm to compute the failure probability of each spreadsheet cell. In the empirical evaluation, we use the Ochiai similarity coefficient, since Ochiai is known to be one of the most efficient similarity coefficients used in SFL techniques [21].

4.2 Spectrum-Enhanced Dynamic Slicing Approach

Spectrum-Enhanced Dynamic Slicing (SENDYS) [4] is a technique that combines SFL with a lightweight model-based software debugging (MBSD) technique [24]. In traditional programming paradigms, similar to SFL, SENDYS uses coverage data and the result of each test execution (pass/fail) of a given SUT as input. In addition, the slices of the negative test cases are required. SENDYS works as follows: the similarity coefficients computed by means of SFL act as a priori probabilities in the MBSD approach. Each statement gets assigned its similarity coefficient as the initial fault probability. The slices of the faulty variables are treated as conflict sets and the minimal hitting sets (i.e. the diagnoses) are computed. A set h is a hitting set for set of conflict sets CO if and only if for all $c \in CO$, there exists a non-empty intersection between c and h (i.e., $\forall c \in CO : c \cap h \neq \emptyset$). From the initial statement fault probabilities, the fault probabilities of the diagnoses are computed. Therefore, the probabilities of the statements contained in the diagnosis are multiplied with the counter-probabilities of the statements not contained in the diagnosis. Afterwards, the probabilities are mapped back to the statements. For each statement, the probabilities of the diagnoses containing that statement are summed up. Finally, the statements are ranked according to their probabilities. The statement with the highest probability is ranked at the first position.

In order to apply Sendys to the spreadsheet paradigm, we propose to make the same modifications as described in the section about the Sfl technique. In addition, we have to use cones instead of slices for the Mbsd part. The major difference between cones and slices are the used dependencies. For slices, control- and data dependencies are used. In contrast, cones only make use of data dependencies.

4.3 Constraint-Based Debugging

There exist several model-based software debugging (Mbsd) techniques which use constraints as part of their debugging strategy, e.g. [14, 25]. In these techniques, program statements are converted into their constraint representation. Each converted statement is connected with a variable representing the health status of the statement: a statement either behaves as specified or the statement has the health status 'abnormal'. A constraint solver is used to compute all possible solutions for the health states of all statements so that the constraints of the program are feasible. All statements with the health status 'abnormal' are explanations for an observed misbehavior. Some solutions of the constraint solver contain several 'abnormal' statements. In this case, all 'abnormal' statements must be changed in order to correct the faulty program. The result of applying a constraint-based debugging technique on a faulty program is a set of 'abnormal' variables representing the health status of the corresponding statements. There is no conclusion which of the statements is more likely to be faulty. Unlike Sfl and Sendys, this method can not generate a likelihood-ranking of possibly faulty statements.

In the context of spreadsheet debugging, cells are used instead of statements: for each cell, the contained formula is converted into a set of constraints. ConBug [5] is a technique that is based on the above described technique, but is designed for debugging spreadsheets.

5 Empirical Evaluation

In this section, we are evaluating the previously described approaches by means of the Euses spreadsheet corpus [6]. In the first part, we are evaluating the ranking of the faulty statements for Sfl and Sendys. In the second part, we are evaluating the size of the result set of ConBug in comparison to the union and intersection of the slices. However, first of all we are going the explain the experimental setup.

In a first filtering step, we skipped around 240 Excel 5.0 spreadsheets that are not compatible with our implementation, since our implementation is build on Apache POI (`http://poi.apache.org/`) and POI does not support Excel 5.0.

In a second filtering step, we removed all spreadsheets containing less than five formulas (about 2,300 files). We have performed this filtering step because automatic fault localization only makes sense for larger spreadsheets. A small spreadsheet is still manageable and thus fault localization can be easily performed manually. For small spreadsheets, a fault correction approach makes more sense than just a fault localization approach. Finally, around 1,400 spreadsheets remain for our case study.

For each spreadsheet, we automatically created up to five first-order mutants. A mutant of a spreadsheet is created by randomly choosing a formula cell of the spreadsheet and applying a mutation operator on it. According to the classification of spreadsheet mutation operators of Abraham and Erwig [26], we used the following mutation operators:

– *Continuous Range Shrinking (CRS):* We randomly choose whether to increment the index of the first column/row or decrement the index of the last column/row in area references.
– *Reference Replacement (RFR):* We randomly choose whether to increment the row or the column index of references. We do not explicitly differentiate between single references and references in non-contiguous ranges. For this, a mutation can change a single reference in a non-contiguous range, but never changes the amount of elements in the range.
– *Arithmetic Operator Replacement (AOR):* We replace '+' with '-' and vice versa and '*' with '/'.
– *Relational Operator Replacement (ROR):* We replace the operators '=', '<', '<=', '>', '>=', and '<>' with one another.
– *Constants Replacement (CRP):*
  - For integer values, we add a random number between 0 and 1000.
  - For real values, we add a random number between 0.0 and 1.0.
  - For Boolean values, we replace 'true' with 'false' and vice versa.
– *Constants for Reference Replacement (CRR):* We replace a reference within a formula through a constant.
– *Formula Replacement with Constant (FRC):* We replace a whole formula with a constant.
– *Formula Function Replacement (FFR):* We replace 'SUM' with 'AVERAGE' and 'COUNT' and vice versa. We replace 'MIN' with 'MAX' and vice versa.

For each mutant, we check whether the following two conditions are satisfied: (1) The mutant must be valid, i.e. it does not contain any circular references. (2) The inserted fault must be revealed, i.e. at least for one output cell, the computed value of the mutant must differ from the value of the original spreadsheet. If one of these conditions is violated, we discard the mutant and generate new mutants until we obtain a mutant that satisfies both conditions. We failed to create mutants for some of the spreadsheets because in many spreadsheets, input values are absent. For this reason, the spreadsheets lack values for output variables. Please note, that the creation of test cases is out of the focus of this paper. We only rely on existing input-output pairs.

We automatically created 622 mutants. Table 1 gives an overview of the characteristic of the created mutants. The number of formulas contained in the spreadsheets ranges from 6 to more than 4,000. This indicates that the evaluated approaches are able to handle large spreadsheets.

Table 1. Characteristics of the created mutants

Characteristic	**Avg**	**Min**	**Max**	**Std.dev**	**Median**
Number of formulas	225.0	6	4,170	384.9	104.5
Number of incorrect output cells	1.7	1	22	1.9	1
Number of correct output cells	64.9	0	2,962	162.5	24

In the first part of the empirical evaluation, we compared the fault localization capabilities of Sfl and Sendys by applying them to the generated mutants. In addition, we contrast these techniques with two primitive techniques, namely the union and intersection of the faulty cones. Table 2 summarizes the results of this comparison. The evaluation was performed on an Intel Core2 Duo processor (2.67 GHz) with 4 GB RAM with Windows 7 Enterprise (64-bit) as operating system and Java 7 as runtime environment. Sendys performs slightly better than Sfl and the intersection of the cones. Since we only created first-order mutants, the intersection of the slices always contains the faulty cell. Please note that in case of higher-order mutants, the faulty cell could be absent in the intersection of the cones. This happens when two independent faults are contained in the same spreadsheet and both faults are revealed by different output cells. Therefore, the intersection of the cones is not the best choice. Concerning the computation time, Sfl has only a small overhead compared to the union and intersection of the cones. Sendys requires nearly five times longer for the computations.

Table 2. Average ranking and computation time of union, intersection, Sfl, and Sendys. The column 'Avg. relative ranking' shows the average ranking of the faulty cell normalized to the number of formula cells per spreadsheet. This evaluation comprises 622 spreadsheets.

Technique	**Avg. absolute ranking**	**Avg. relative ranking**	**Avg. comp. time (in ms)**
Union (cones of faulty output)	41.1	27.3 %	15.6
Intersection (cones of faulty output)	30.8	22.0 %	15.6
Sfl	26.3	20.3 %	16.9
Sendys	24.3	19.7 %	79.6

Figure 1 graphically compares the fault localization capabilities of the approaches for the 622 investigated faulty program versions. The x-axis represents the percentage of formula cells that is investigated. The y-axis represents the percentage of faults that are localized within that amount of cells. This figure reads as follows: if you investigate the top 40 % ranked cells in all of the 622 faulty spreadsheets, SFL and SENDYS find the fault in 80 % of the Spreadsheets. It can be seen that SFL and SENDYS perform slightly better than the intersection and marginally better than the union of the cones. This means that faults can be detected earlier than when using the intersection or the union.

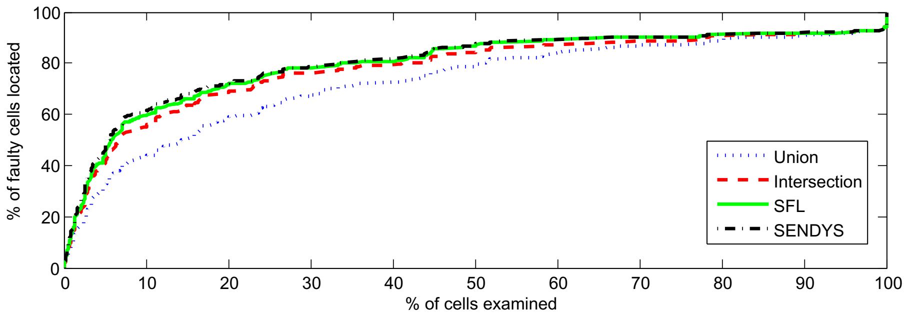

Fig. 1. Comparison of the SFL, SENDYS, the Union and Intersection of the cones in terms of the amount of formula cells that must be investigated

In the second part of the empirical evaluation, we investigate the debugging capabilities of CONBUG. We separated the evaluation of CONBUG from the evaluation of SFL and SENDYS because the prototype implementation of CONBUG does not support all available mathematical operations available in Excel. Therefore, we filter out all spreadsheets which contain unsupported operations. Subsequently, 227 mutants remain for the evaluation of CONBUG. These mutants contain on average 219.8 formula cells. The largest mutant contains 2564 formula cells. Table 3 compares CONBUG to the union and the intersection of the cones. For completeness reasons, we add the data of SFL and SENDYS for the 227 spreadsheets. CONBUG performs better than the union, but worse than the intersection. However, CONBUG guarantees to contain the faulty cell even in the

Table 3. Average ranking and computation time of union, intersection, SFL, SENDYS, and CONBUG. The column 'Avg. relative ranking' shows the average ranking of the faulty cell normalized to the number of formula cells per spreadsheet. This evaluation comprises 227 spreadsheets.

Technique	**Avg. absolute ranking**	**Avg. relative ranking**	**Avg. comp. time (in ms)**
Union (cones of faulty output)	34.8	29.3 %	14.0
Intersection (cones of faulty output)	33.6	27.5 %	13.9
SFL	32.9	27.1 %	15.0
SENDYS	31.9	27.0 %	63.9
CONBUG	33.9	27.9 %	631.7

case of multiple faults. However, the faulty cell can be absent in the intersection of the cones. From Table 3 we see that the differences of the obtained results are small and might not be statistical significant.

Why are the results of Table 3 so close? One explanation might be the structure of the used spreadsheets. In particular, the 227 spreadsheets used for obtaining the results of Table 3 have 1.2 faulty output variables on average whereas the remaining 395 spreadsheets used in Table 2 have on average 2 faulty output variables. The low number of faulty output variables might be a reason for poor performance of ConBug. Further investigations are necessary to clarify the relationship of the structure of the spreadsheets and the performance of the approaches. Moreover, the debugging performance in case of multiple faults in spreadsheets is also an open research question. We expect better results for ConBug in case of multiple faults, similar as in constraint-based approaches for traditional programming paradigms [27].

6 Conclusion

While spreadsheets are used by a considerable number of people, there is little support for automatic spreadsheet debugging. In this paper, we addressed this gap. In particular, we adapted and applied to spreadsheets three popular debugging techniques designed for more traditional procedural or object-oriented programming languages. To this end, we formally defined the basic elements of spreadsheets and formalized fault localization in spreadsheets as spreadsheet debugging problem. In addition, we explained what modifications to the traditional debugging techniques are necessary. The main modification is to use cones instead of execution traces and slices.

We evaluated the fault localization capabilities of the proposed techniques, Sfl, Sendys, and ConBug, using the well-known Euses spreadsheet corpus [6]. The evaluation showed that Sfl and Sendys are the most promising techniques. However, the evaluation needs to be extended in several aspects: (1) It is necessary to evaluate higher-order mutants. (2) The discussed techniques are only a small selection of the available traditional debugging techniques. Thus, other debugging techniques should be adapted to spreadsheets. We plan to make the mutants used in this evaluation as well as higher-order mutants publicly available. This will ensure that new spreadsheet debugging techniques can be compared to the techniques discussed in this paper. Furthermore, the acceptance of such debugging techniques must be evaluated throw a user study.

Sfl and Sendys are debugging techniques which rank cells according to their likelihood of containing the fault. In contrast, ConBug is a debugging technique that filters out cells which cannot explain the observed faulty values. Therefore, ConBug can be used to filter out statements from the rankings of Sfl and Sendys. We plan to evaluate this filter mechanism in future work.

Acknowledgments. This work was supported by the Foundation for Science and Technology (FCT), of the Portuguese Ministry of Science, Technology, and Higher Education (MCTES), under Project PTDC/EIA-CCO/108613/2008, and the competence network Softnet Austria II (www.soft-net.at, COMET K-Projekt) funded by the Austrian Federal Ministry of Economy, Family and Youth (bmwfj), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

References

1. Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S.: The state of the art in end-user software engineering. ACM Computing Surveys (2011)
2. Chadwick, D., Knight, B., Rajalingham, K.: Quality control in spreadsheets: A visual approach using color codings to reduce errors in formulae. Software Quality Journal 9(2), 133–143 (2001)
3. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, TAICPART-MUTATION 2007, pp. 89–98. IEEE Computer Society, Washington, DC (2007)
4. Hofer, B., Wotawa, F.: Spectrum enhanced dynamic slicing for better fault localization. In: Proceedings of the European Conference on Artificial Intelligence (ECAI 2012). Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 420–425. IOS Press (2012)
5. Abreu, R., Riboira, A., Wotawa, F.: Constraint-based debugging of spreadsheets. In: Proceedings of the 15th Ibero-American Conference on Software Engineering (2012)
6. Fisher, M.I., Rothermel, G.: The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In: 1st Workshop on End-User Software Engineering, pp. 47–51 (2005)
7. Burnett, M.M., Cook, C.R., Pendse, O., Rothermel, G., Summet, J., Wallace, C.S.: End-user software engineering with assertions in the spreadsheet paradigm. In: Clarke, L.A., Dillon, L., Tichy, W.F. (eds.) ICSE, pp. 93–105. IEEE Computer Society (2003)
8. Panko, R.R.: Recommended practices for spreadsheet testing. CoRR abs/0712.0109 (2007)
9. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Mdsheet: A framework for model-driven spreadsheet engineering. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) ICSE, pp. 1395–1398. IEEE (2012)
10. Ruthruff, J., Creswick, E., Burnett, M., Cook, C., Prabhakararao, S., Fisher II, M., Main, M.: End-user software visualizations for fault localization. In: Proceedings of the 2003 ACM Symposium on Software Visualization, SoftVis 2003, pp. 123–132. ACM, New York (2003)
11. Ayalew, Y., Mittermeir, R.: Spreadsheet debugging. Bilding Better Business Spreadsheets - from the ad-hoc to the quality-engineered. In: Proceedings of EuSpRIG 2003, Dublin, Ireland, July 24-25, pp. 67–79 (2003)
12. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1), 57–95 (1987)

13. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In: Proc. ASE 2009. IEEE CS (2009)
14. Wotawa, F., Weber, J., Nica, M., Ceballos, R.: On the Complexity of Program Debugging Using Constraints for Modeling the Program's Syntax and Semantics. In: Meseguer, P., Mandow, L., Gasca, R.M. (eds.) CAEPIA 2009. LNCS, vol. 5988, pp. 22–31. Springer, Heidelberg (2010)
15. Jannach, D., Engler, U.: Toward model-based debugging of spreadsheet programs. In: 9th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2010), Kaunas, Lithuania, August 25-27, pp. 252–264 (2010)
16. Abraham, R., Erwig, M.: Goaldebug: A spreadsheet debugger for end users. In: 29th IEEE International Conference on Software Engineering, pp. 251–260 (2007)
17. Abraham, R., Erwig, M.: Goal-directed debugging of spreadsheets. In: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC 2005, pp. 37–44. IEEE Computer Society, Washington, DC (2005)
18. Rothermel, K.J., Cook, C.R., Burnett, M.M., Schonfeld, J., Green, T.R.G., Rothermel, G.: WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation. In: Proc. ICSE 2000, pp. 230–239. ACM (2000)
19. Cunha, J., Fernandes, J.P., Ribeiro, H., Saraiva, J.: Towards a Catalog of Spreadsheet Smells. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part IV. LNCS, vol. 7336, pp. 202–216. Springer, Heidelberg (2012)
20. Tikir, M.M., Hollingsworth, J.K.: Efficient instrumentation for code coverage testing. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2002, pp. 86–96. ACM, New York (2002)
21. Abreu, R., Zoeteweij, P., van Gemund, A.: An evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006, pp. 39–46. IEEE Computer Society, Washington, DC (2006)
22. Janssen, T., Abreu, R., van Gemund, A.: Zoltar: A toolset for automatic fault localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, pp. 662–664. IEEE Computer Society, Washington, DC (2009)
23. Campos, J., Riboira, A., Perez, A., Abreu, R.: Gzoltar: an eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pp. 378–381. ACM, New York (2012)
24. Wotawa, F.: Bridging the gap between slicing and model-based diagnosis. In: Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE 2008), pp. 836–841 (2008)
25. Wotawa, F., Nica, M.: On the compilation of programs into their equivalent constraint representation. Informatica (Slovenia) 32(4), 359–371 (2008)
26. Abraham, R., Erwig, M.: Mutation operators for spreadsheets. IEEE Transactions on Software Engineering, 94–108 (2009)
27. Hofer, B., Wotawa, F.: On the empirical evaluation of fault localization techniques for spreadsheets. In: 12th International Conference on Quality Software, pp. 41–48 (2012)