Elisabeth Getzner, BSc

Improvements for Spectrum-based Fault Localization in Spreadsheets

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisors

Univ.-Prof. Dipl-Ing. Dr.techn. Franz Wotawa,

Dipl.-Ing. Dr.techn. Birgit Hofer

Institute for Software Technology

Graz, May 2015

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the used sources. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, am _____ _____

Datum Unterschrift

Abstract

Spreadsheets are a popular tool often used in critical computations. Research has shown that a large percentage of professionally used spreadsheets contain errors, potentially leading to great financial losses. As a result, interest in quality assurance for spreadsheets has increased, leading to the adaptation and creation of techniques to avoid, find and repair errors in spreadsheets. One such technique is fault localization, which aids in spreadsheet debugging provided one or more cells produce an unexpected output. Fault localization uses this information to produce a set of cells that contribute to this output and may contain the fault.

In particular, spectrum-based fault localization, short SFL, ranks cells so that the highest ranked cell has the highest fault likelihood. The effectiveness of SFL is considerably impaired if two or more cells are equally likely to be faulty, forming a tie. In this thesis, we propose several improvements for SFL that either reduce the number of cells that need inspection or create additional prioritization, helping end-users to debug their spreadsheets more efficiently.

One promising improvement is Grouping, which clusters similar cells in a spreadsheet to account for the fact that many spreadsheet users copy and paste large parts of the spreadsheet. The size of the tie is reduced as grouped cells form a single unit, even though the actual number of cells in the ranking remains constant. Another possibility to improve the ranking is to reduce the input passed to SFL, resulting in a reduction of the search space. For this purpose, we propose the Blocking and Dynamic Slicing algorithms, which show moderate improvements on spreadsheets with specific structural requirements.

Tie-breaking, as opposed to input or tie reduction, focuses on increasing prioritization by breaking a set of tied cells. We propose a number of tie-breaking strategies based on position, distance and formula metrics. These strategies aim to increase the fault likelihood for suspicious cells without reducing the number of cells in the ranking. The effectiveness of tie-breaking strategies is typically dependent on the structure of the spreadsheets. We found that distance-based strategies are the most effective at prioritizing authentic faults.

We evaluate the proposed techniques using three spreadsheet corpora, two of which were created as part of this thesis. These corpora allow for a more detailed evaluation, as each offers unique structural properties. The most significant improvement over previous spreadsheet corpora is the existence of realistic testing decisions in one corpus and the existence of authentic faults in another, providing new opportunities for the evaluation of future fault localization techniques for spreadsheets.

Kurzfassung

Spreadsheets oder Tabellenkalkulationen werden häufig für kritische Berechnungen verwendet. Wissenschaftliche Untersuchungen zeigen jedoch, dass viele der professionell verwendeten Spreadsheets Fehler enthalten, was zu hohen finanziellen Verlusten führen kann. Folglich hat sich das Interesse für Qualitätssicherung für Spreadsheets erhöht und eine Vielzahl an Techniken wurden entwickelt oder adaptiert, um Fehler zu vermeiden, zu finden oder auszubessern. Einer dieser Ansätze ist die Fehlerlokalisierung, welche die Fehlerbehebung unterstützt wenn eine oder mehrere Zellen einen unerwarteten Wert liefern. Diese Information wird verwendet um eine Menge von Zellen zu identifizieren, die zu dem unerwarteten Wert beitragen und daher den Fehler beinhalten können.

Wir betrachten Spektrum-basierende Fehlerlokalisierung, kurz SFL, welche darauf abzielt die Zellen so zu reihen, dass die Zelle mit der höchsten Fehlerwahrscheinlichkeit die höchste Priorität besitzt. Die Wirksamkeit von SFL ist stark beeinträchtigt wenn zwei oder mehr Zellen dieselbe Fehlerwahrscheinlichkeit aufzeigen und nicht weiter priorisiert werden können. In diesem Fall sprechen wir von einem Tie oder Gleichstand. In dieser Arbeit stellen wir mehrere Verbesserungsvorschläge für SFL vor, die entweder die Anzahl der zu untersuchenden Zellen reduzieren oder zusätzliche Priorisierung liefern und so dem Benutzer dabei helfen, Fehler in Spreadsheets effizienter zu beseitigen.

Ein erfolgversprechender Ansatz ist Grouping, welches ähnliche Zellen gruppiert und somit Rücksicht auf das häufige Kopieren von Elementen in Spreadsheets nimmt. Während die Anzahl der Zellen, die einen Tie formen, unverändert bleibt, ist die Größe des Ties reduziert, da gruppierte Zellen eine Einheit darstellen. Eine weitere Verbesserungsmöglichkeit ist es, den gesamten Suchraum für SFL zu reduzieren. Zu diesem Zweck schlagen wir Blocking und Dynamic Slicing Algorithmen vor, die moderate Verbesserungen bei Spreadsheets mit spezifischen strukturellen Anforderungen zeigen.

Ein weiterer Ansatz ist das Tie-Breaking, welches versucht, die Priorisierung innerhalb eines Ties zu verbessern. Wir stellen einige solcher Strategien vor, die auf Positionen, Distanzen oder Formelmetriken einzelner Zellen basieren. Diese Strategien sollen die Fehlerwahrscheinlichkeit von auffälligen Zellen erhöhen, ohne die Anzahl der Zellen im Ranking zu reduzieren. Die Effektivität dieser Tie-Breaking Strategien ist abhängig von der Struktur des Spreadsheets. Unsere Auswertung zeigt, dass Distanz-basierte Strategien am effektivsten sind, um authentische Fehler zu priorisieren.

Wir evaluieren die vorgeschlagenen Techniken anhand von drei Testkorpora, wobei zwei davon in dieser Arbeit erstellt wurden. Diese Testdaten erlauben eine detaillierte Auswertung, da jeder einzigartige strukturelle Eigenschaften besitzt. Die bedeutendsten Verbesserungen gegenüber bisherigen Testdaten sind realistische Anwendereingaben und authentische Fehler. Diese Daten bieten neue Möglichkeiten für die Evaluierung zukünftiger Fehlerlokalisierungstechniken.

Acknowledgements

I would like to thank first and foremost my advisor Birgit Hofer, who spent hours reading and discussing my thesis, guiding me in the right direction and offering advice. I am equally grateful for the continuous encouragement expressed by my supervisor Professor Franz Wotawa, who also provided the necessary facilities for our research.

To Professor Margaret Burnett of the Oregon State University, I am sincerely grateful for the experiment data on which we were able to base a spreadsheet testing corpus. For the spreadsheet corpus based on student submissions, I thank Salvatore Valeskini for providing us with the spreadsheets and permitting the publication of the resulting corpus.

Finally, I wish to thank Patrick Koch and Dominic Amann, who kindly proof-read the thesis and offered helpful comments and ideas. I also place on record my gratitude to my friends and family, who supported me throughout my studies.

Contents

List of Figures

List of Tables

1 Introduction

Spreadsheets are one of the most popular and widely used parts of office suites around the world. Spreadsheet environments such as Microsoft Excel, LibreOffice Calc or Google Sheets offer a straight-forward interface to handle a wide variety of table-based data. Usage varies from small personal spreadsheets that contain little to no formulas to highly complex professional spreadsheets that compute data critical to the success or financial gain of companies.

Even though values computed in spreadsheets influence critical corporate decisions, the lack of peer review and coding support as well as the basic structure of spreadsheet environments lead to many spreadsheets containing faults. Panko [24] estimates the number of faulty in-production spreadsheets to be around 88 %. The European Spreadsheet Risk Interest Group, short EuSpRIG, has a collection of horror stories[1] detailing critical spreadsheet errors, with the most recent one from 2013. The largest bank in the United States, JPMorgan Chase, suffered trading losses of 6.2 billion dollars due to the risks taken by a trader known as "the London Whale". An investigation revealed that spreadsheet errors were partially to blame, as the computed Value-at-Risk model was erroneous. After subtracting the old rate from the new one, the value was divided by the sum instead of the average of the two values [20]. This minor mistake lead to the estimated risk being lowered by a factor of two and ultimately to the loss of billions of dollars.

Recent research has therefore focused on quality assurance in spreadsheets, supporting users in spreadsheet creation and aiding users in debugging faulty spreadsheets. Jannach *et al.* [19] provide an extensive overview of such methods, including fault localization, which is the focus of this thesis. Fault localization helps to debug a faulty spreadsheet if the user can identify one or more cells that produce an unexpected value. In this case, fault localization provides a selection of cells that are likely to cause the erroneous behavior.

To provide a simple example, let us assume we wish to compute the salaries of our employees based on their work hours as well as bonuses for those that were particularly diligent. Example 1.1 shows such a simplified salary calculation, assigning a bonus if the employee worked more than 15 hours. A fault occurred in cell `D2`, indicated by the red dashed border, where the wrong cell is used to compute the size of the bonus for Jones. Faults may occur in constants, such as the entered work hours, as well as the formulas used to compute the salary and bonuses. We assume that, typically, the fault lies in formula cells, as they are more complex than constant cells. To determine whether the spreadsheet is faulty, we can use values from previous years or estimates. For example,

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

we may know that the total amount computed in E5 is unexpected. Fault localization techniques assist in debugging by highlighting cells that contribute to the erroneous cell E5 and possibly ranking cells with higher fault likelihood, thereby assisting in efficient debugging. For this example, fault localization may exclude C5 and D5 from the search space, as they do not influence the result in E5. The other cells containing formulas, including our faulty cell D2, are highlighted.

	A	B	C	D	E
1		Hours	Salary	Bonus	Sum
2	Jones	17	272	26	298
3	Smith	13	208	0	208
4	Rogers	20	320	40	360
5	Total		800	66	866

(a) The value view of the spreadsheet, where the value in the faulty cell D2 should be 34 instead of 26.

	A	B	C	D	E
1		Hours	Salary	Bonus	Sum
2	Jones	17	=B2*16	=IF(B2>15; C3 /8;0)	=SUM(C2:D2)
3	Smith	13	=B3*16	=IF(B3>15; C3 /8;0)	=SUM(C3:D3)
4	Rogers	20	=B4*16	=IF(B4>15; C4 /8;0)	=SUM(C4:D4)
5	Total		=SUM(C2:C4)	=SUM(D2:D4)	=SUM(E2:E4)

(b) The formula view, where the faulty cell D2 references B2 (blue border) and C3 (red border). Instead of Smith's salary in C3, the salary in C2 should be referenced for the Bonus calculation for Jones.

Example 1.1: A spreadsheet of a bonus calculation with the fault in D2 highlighted with a red dashed border.

The number of fault localization techniques for spreadsheets is steadily growing, with well-known algorithms for traditional programming being applied to the spreadsheet context [15, 5]. These approaches are typically separated in trace-based approaches, which analyze cell dependencies and return a ranking, and model-based approaches, which provide more exact fault explanations in the form of diagnoses, but often lack prioritization. In this thesis, we focus on improving spectrum-based fault localization, short SFL, which is a computationally inexpensive trace-based approach that ranks cells according to their fault likelihood, thereby not only limiting the search space but also prioritizing the user's search.

Existing research [5, 27] has tended to focus on improving fault localization by limiting the number of cells that need inspection. Many approaches, both trace- and model-based, work well at reducing the search space or ranking the faulty cell at the highest likelihood. Previous work in this area [15] has shown that a key issue in fault localization is the fact that several cells may have the same likelihood to contain the fault, leaving the user to decide which cells to inspect first. In this thesis, we propose several strategies to improve existing fault localization techniques, including approaches that reduce the search space

as well as strategies to improve prioritization for such tied statements.

Grouping [23] is a strategy that reduces the complexity of the debugging problem as it clusters cells with similar or identical formulas to a single unit. Blocking [27] reduces the search space by ranking cells that are unlikely to contain the fault significantly lower than SFL alone. Dynamic Slicing [30] removes cells from the search space which do not influence the erroneous cells by analyzing conditionals. The tie-breaking strategies are separated in position- and metric-based tie-breaking. Position-based tie-breaking is based on research by Xu *et al.* [32], who propose similar strategies for the programming language C. Metric-based tie-breaking uses formula metrics or code-smells for spreadsheets, which have been the topic of some research [12, 7] but whose effects on fault localization have not been previously examined. To evaluate these approaches, we introduce two additional spreadsheet corpora which alleviate some of the issues with existing test data.

We show that the grouping of similar cells in spreadsheets considerably improves the perceived position of the faulty cell in the ranking, while also reducing the complexity of the search space. Input reduction and tie-breaking strategies are discussed and evaluated, and distance-based tie-breaking strategies are found to be the most successful at increasing prioritization.

The main contributions of this thesis are:

- a description of existing issues with SFL, such as handling of multiple faults, the existence of large ties and low rank of the faulty cell,
- the development of Grouping strategies, which cluster similar formula cells to one unit so that the ranking better reflects the model of the spreadsheet,
- the adaptation of two strategies, Blocking and Dynamic Slicing, which further reduce the search space,
- the selection and implementation of position- and metric-based tie-breaking strategies to increase prioritization of the ranking,
- the conversion of test data from a user study [27] to create a spreadsheet corpus with authentic user input,
- the creation of a spreadsheet testing corpus with authentic faults based on student submissions of an Excel course,
- a detailed analysis and comparison of the three used spreadsheet corpora and
- the evaluation of the proposed improvements on the basis of the three corpora.

This thesis is structured as follows: Chapter 2 defines the basic terminology for spreadsheets and fault localization. Section 2.1 defines spreadsheets, the spreadsheet language and other terminology used in this work. Section 2.2 explains fault localization for

spreadsheets using the spectrum-based fault localization technique developed in our previous work [15]. Common terms for fault localization are defined and the metrics to measure the effectiveness of our techniques are introduced.

We examine related work in Chapter 3, briefly summarizing existing fault localization techniques. We review model- and trace-based approaches, where model-based approaches focus on reducing the search space and offer precise diagnoses and trace-based approaches create a ranking of likely faulty cells. We also touch on related approaches such as fault detection and fault repair techniques.

Chapter 4 lists a number of problems that exist in contemporary fault localization techniques. Section 4.1 outlines the issues multiple faults present to fault localization techniques. Section 4.2 discusses the difficulty of providing correct and useful test cases for the spreadsheet, focusing on user mistakes when providing information and cells whose values are coincidentally correct. Finally, we examine the issue of many critically tied statements in Section 4.3.

We propose several improvements to fault localization, which we describe in detail in Chapter 5. Section 5.1 discusses the idea to group similar cells to a single unit. Section 5.2 shows how two existing techniques, Blocking and Dynamic Slicing, can function as input reduction strategies. Finally, Section 5.3 selects several tie-breaking strategies and metrics for spreadsheets and shows how these metrics can split up and further prioritize a number of critically tied cells.

Chapter 6 describes the properties and creation of the three spreadsheet corpora used in the evaluation. Section 6.1 discusses the Euses corpus [10] used in previous evaluations [15]. Section 6.2 introduces a new spreadsheet corpus based on student submissions containing authentic faults. Section 6.3 describes the conversion of test data from a user study by Ruthruff *et al.* [27] to our local format. This testing corpus contains injected faults but authentic user input. Finally, the corpora are compared in detail in Section 6.4.

We evaluate the proposed improvements on all three corpora and discuss the results of the evaluation in Chapter 7. This chapter is split into two parts, as we first analyze the performance of Sfl alone to motivate the need for improvement in Section 7.1. The second part is the discussion of the evaluated data in Section 7.2, comparing the approaches to one another and highlighting our findings.

We conclude this thesis in Chapter 8, which summarizes our findings and identifies potential areas of future work.

2 Preliminaries

This chapter establishes definitions required throughout the rest of this thesis. We define common terminology for spreadsheets in Section 2.1 and the basics for spectrum-based fault localization in Section 2.2.

2.1 Spreadsheet Terminology

Let us define spreadsheets and their related terms, following closely the definitions from our previous work [15]. A spreadsheet is a collection of cells in a two dimensional matrix, where each cell can be identified by its unique position within the spreadsheet given by its row and column identifier. Formally speaking, we define the functions $\varphi_x(c)$ and $\varphi_y(c)$ to return the numerical value for the column and row of a cell c respectively.

A single cell $c \in \textsc{Cells}$ has a formula $\ell(c)$ and a value $\nu(c)$, where $\ell(c)$ is an expression in the language $\mathcal{L}$. The value $\nu(c)$ is the evaluation of the expression $\ell(c)$, which is either undefined ϵ, $\perp$ if the evaluation leads to an error or otherwise any number, Boolean or string value. If the expression is empty or undefined, the value ϵ is returned. The value $\nu(c)$ can also be equated to the visible result often presented by the spreadsheet environment, for example Microsoft Excel.

We now define the language $\mathcal{L}$, which is a non-recursive functional language consisting of a collection of function definitions. For spreadsheets, each cell contains one such non-recursive function and can therefore have exactly one value. In the interest of brevity, we only define the elements of $\mathcal{L}$ that are needed to reconstruct later examples.

Definition 2.1.1. (*Syntax of* $\mathcal{L}$) The syntax of $\mathcal{L}$ is defined recursively so that:

- Numbers, Booleans or strings as well as ϵ are constants $\kappa \in \mathcal{L}$.
- Cell identifiers are elements of $\mathcal{L}$, i.e. $\textsc{Cells} \subset \mathcal{L}$.
- Let the expressions $e_1, e_2, e_3 \in \mathcal{L}$, it holds that
  - $\texttt{(}e_1\texttt{)}$ is an element of $\mathcal{L}$,
  - $e_1 \circ e_2$, given a binary operator $\circ \in \{+, -, *, /, <, >, =\}$, is an element of $\mathcal{L}$,
  - $\texttt{IF(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}$ is also an element of $\mathcal{L}$.
- The expression $\texttt{SUM(}c_1\texttt{:}c_2\texttt{)}$ is an element of $\mathcal{L}$, where $c_1\texttt{:}c_2$ is an area of cells which, given that both coordinates of c_1 are smaller than or equal to the coordinates of c_2, describes the following collection of cells:

$$c_1\texttt{:}c_2 := \left\{ c_i \in \textsc{Cells} \;\middle|\; \begin{array}{c} \varphi_x(c_1) \leq \varphi_x(c_i) \leq \varphi_x(c_2) \wedge \\ \varphi_y(c_1) \leq \varphi_y(c_i) \leq \varphi_y(c_2) \end{array} \right\}.$$

The syntax of $\mathcal{L}$ can easily be extended to allow unary and additional binary operators as well as defining additional functions in a similar fashion to `SUM`.

For the semantics of $\mathcal{L}$, we define an interpretation function $[\![e]\!]$ which maps the expression $e \in \mathcal{L}$ to a value which may either be ϵ, $\perp$ or a number, Boolean or string, similarly to the definitions above.

Definition 2.1.2. (*Semantics of* $\mathcal{L}$) Given an expression $e \in \mathcal{L}$ and a function ν mapping a cell identifier to a value, the following semantics apply:

- If e is a constant κ, the value of the constant is used, i.e. $[\![e]\!] = \kappa$.
- If e is a cell identifier for cell c then $[\![e]\!] = \nu(c)$, unless the returned value is ϵ, in which case a number of zero is assumed.

$$[\![e]\!] = \begin{cases} 0 & \text{if } \nu(c) = \epsilon \\ \nu(c) & \text{otherwise} \end{cases}$$

- If e has the form $\texttt{(}e_1\texttt{)}$ it holds that $[\![e]\!] = [\![e_1]\!]$.
- If e has the form $e_1 \circ e_2$ and $\circ \in \{+, -, *, /, <, >, =\}$, it holds that

$$[\![e_1 \circ e_2]\!] = \begin{cases} [\![e_1]\!] \circ [\![e_2]\!] & \text{if } e_1 \text{ and } e_2 \text{ evaluate to numbers} \\ \perp & \text{otherwise} \end{cases}$$

  This means that if any sub-expression produces an error or is empty, an error is returned. Additionally, operations are only allowed for numbers, while operations on Booleans or strings are not defined.

- If e has the form $\texttt{IF(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}$, the evaluation is as follows:

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

- If e has the form $\texttt{SUM(}c_1\texttt{:}c_2\texttt{)}$, the evaluation is as follows:

$$[\![e]\!] = \begin{cases} \sum_{c_i \in c_1:c_2} [\![c_i]\!] & \text{if all cells evaluate to numbers} \\ \perp & \text{otherwise} \end{cases}$$

To further aid the understanding of spreadsheets, we define the term of "cell reference" more closely, as it is a key point in all subsequent fault localization algorithms.

Definition 2.1.3. (*Cell Reference*) An expression $e \in \mathcal{L}$ references a cell c if it contains the cell identifier of c.

Furthermore, we define the function $\rho(e)$ to get all cell references occurring in the expression e.

Definition 2.1.4. (*The function* ρ) This function returns the set of all cells that are referenced by expression $e \in \mathcal{L}$, which can be defined recursively so that:

- If e is a constant κ then $\rho(e) = \varnothing$.
- If e is a cell identifier for cell c then $\rho(e) = \{c\}$.
- If e has the form $\texttt{(}e_1\texttt{)}$ then $\rho(e) = \rho(e_1)$.
- If e has the form $e_1 \circ e_2$ then $\rho(e) = \rho(e_1) \cup \rho(e_2)$.
- If e has the form $\texttt{IF(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}$ then $\rho(e) = \rho(e_1) \cup \rho(e_2) \cup \rho(e_3)$.
- If e has the form $\texttt{SUM(}c_1\texttt{:}c_2\texttt{)}$ then $\rho(e) = c_1\texttt{:}c_2$.

The above definitions allow us to define spreadsheets more formally, being a matrix of cells with values and expressions, so that $\forall c \in \textsc{Cells} : \nu(c) = [\![\ell(c)]\!]$. For the recursive definitions we used above, we need to introduce additional restrictions to spreadsheets so that these functions terminate. First, we need to define the set of cells Cells as finite. Second, we need to ensure that there are no cyclic references in the expression, for example a cell referencing itself. We handle the first issue by defining a spreadsheet that contains a finite number of cells.

Definition 2.1.5. (*Finite Spreadsheet*) We have a finite subset of cells, $\Pi \subseteq \textsc{Cells}$, where each cell has a non-empty expression or is referenced by another cell, so that $\forall c \in \Pi : (\ell(c) \neq \epsilon \vee \exists c' \in \Pi : c \in \rho(\ell(c')))$

We handle the second issue of cyclic references by creating a graph from the spreadsheet. At first, let us define direct dependence between two cells.

Definition 2.1.6. (*Direct Dependence*) Given two cells $c_i, c_j \in \Pi$, c_i is said to directly depend on c_j if and only if c_i references c_j in its expression, i.e. $(c_i \rightharpoonup c_j) \leftrightarrow c_j \in \rho(\ell(c_i))$.

From these direct dependencies, we can now form a data dependence graph for each cell reference.

Definition 2.1.7. (*Data Dependence Graph (DDG)*) The data dependence graph or DDG of a spreadsheet Π is a tuple (V, A) where:

- V is the set of vertices or nodes of the graph, with a node n_c for each cell $c \in \Pi$ and
- A is the set of directed edges or arcs, where the arc (n_{c_i}, n_{c_j}) exists if and only if there exists a direct dependence between the cells c_i and c_j, meaning

$$A = \left\{(n_{c_i}, n_{c_j}) \mid n_{c_i}, n_{c_j} \in V \wedge (c_i \rightharpoonup c_j)\right\}.$$

This graph allows a more general definition of dependence, where cell c_2 is said to depend on c_1 if there exists a path from n_{c_2} to n_{c_1}, allowing for both direct and indirect dependence. While the cyclic references in the form of self reference can be detected easily, to make the spreadsheet feasible we need to restrict it in a way that does not allow any circles in the DDG. We therefore define a feasible spreadsheet as follows.

Definition 2.1.8. (*Feasible Spreadsheet*) A spreadsheet $\Pi \subseteq \textsc{Cells}$ is said to be feasible if and only if the resulting DDG is acyclic.

We have now formally stated our expectations for the given spreadsheets, and in the future we assume that any spreadsheet given for fault localization is finite and feasible. In the following paragraphs, we explain some common terminology needed for spreadsheets and fault localization with the aid of Example 1.1.

Cell Identifiers

To identify single cells, it is common to use a series of letters starting with `A` for columns and numbers starting with `1` for rows. Therefore, the first cell in a spreadsheet has the coordinates `A1` and can be identified as such within the spreadsheet. As this is the terminology common in spreadsheet environments, we also use this notation for the examples in this thesis, forgoing the more formal definition of $\varphi_x(c)$ and $\varphi_y(c)$ for columns and rows respectively.

Spreadsheets can also be collected by workbooks, which may contain one or more spreadsheets. These spread- or worksheets are given names, so that cell `A1` in spreadsheet `Bonus` can be identified from anywhere within the workbook as `Bonus!A1`. To simplify our explanations, our examples assume a workbook that includes only a single worksheet, where two coordinates suffice to identify a cell. This simplification refers only to our descriptions, the implemented tools naturally support references to different worksheets.

Input / Result

Spreadsheet programming differs from traditional, procedural programming in many areas. There are no function definitions in spreadsheets, therefore no clear input parameters and return values are defined. The concept of variables is also different: A cell can be compared to a variable c, yet it can only be defined once in its expression $\ell(c)$ and the produced value $\nu(c)$. We distinguish between constant cells and formula cells, where a formula cell references at least one cell, i.e. $\rho(\ell(c)) \neq \varnothing$. We define input and result cells as follows:

Definition 2.1.9. (*Input, Result*) In spreadsheets, input cells are cells that are referenced in other cells, but do not reference any cells. Result cells are formula cells that reference cells but are not referenced by any cells themselves. Formally we state for cell c that

$$c \in \textsc{Input} : \rho(\ell(c)) = \varnothing \ \wedge \ \exists c' \in \textsc{Cells} : c \in \rho(\ell(c')) \text{ and}$$
$$c \in \textsc{Result} : \rho(\ell(c)) \neq \varnothing \ \wedge \ \nexists c' \in \textsc{Cells} : c \in \rho(\ell(c')).$$

In Example 1.1, input cells are displayed with blue font color (`B2:B4`) and results cells are in purple font (`C5:E5`). Note that this definition allows cells with a complex formula (i.e. $\ell(c) \neq \kappa$) that do not reference any other cells to also be considered input cells. For fault localization, input cells are often assumed to be correct. In this case, some errors, for example by changing constants in a formula, could not be found with this definition of input cells.

Fault

The terms *fault*, *error* and *failure* are often used interchangeably in literature. To avoid confusion, in this paper we refer to the reason for unexpected behavior as a fault or a true fault, whereas a symptom of the fault (i.e. unexpected output) is described as an erroneous output. The spreadsheet in Example 1.1 is faulty, which could be recognized by the erroneous output produced for the total sum of salaries in `E5`. The fault is revealed in Figure 1.1b, where the bonus calculation in `D2` wrongly references the salary of a different employee (`C3`) instead of the same (`C2`). If this fault were not present, the cell value of the bonus in `D2` would be 34, and the sum in `E5` would be 874.

Multiple Faults

Only a single fault was injected in the spreadsheet in Example 1.1, as some approaches can only debug single faults and to keep the complexity of the example low. However, spreadsheets may contain more than one fault, leading to a multiple fault complexity of the debugging problem. Example 2.1 shows how multiple faults can easily occur, in this case by copying the fault from `D2` down to `D3` and `D4`.

	A	B	C	D	E
1		Hours	Salary	Bonus	Sum
2	Jones	17	272	26	298
3	Smith	13	208	0	208
4	Rogers	20	320	100	420
5	Total		800	***126***	***926***

(a) The value view of the spreadsheet, with the faults in `D2:D4`.

	A	B	C	D	E
1		Hours	Salary	Bonus	Sum
2	Jones	17	=B2*16	=IF(B2>15; C3 /8;0)	=SUM(C2:D2)
3	Smith	13	=B3*16	=IF(B3>15; C4 /8;0)	=SUM(C3:D3)
4	Rogers	20	=B4*16	=IF(B4>15; C5 /8;0)	=SUM(C4:D4)
5	Total		=SUM(C2:C4)	***=SUM(D2:D4)***	***=SUM(E2:E4)***

(b) The formula view, where the cell references are shifted by one row erroneously.

Example 2.1: A spreadsheet of a bonus calculation with three faults, with the faulty references in red and the faulty cells highlighted with a red dashed border. Note that `C5` is no longer a result cell, as it is now referenced by `D4`.

The faulty cell references in column `D` use cells from the row below their own and are in red font, with the red border indicating the wrongly referenced cells. Note that `C5` is no longer a result cell, as it is now erroneously referenced by cell `D4`. This example is useful to explain multi-fault concepts in more detail.

Dependency Graphs

To facilitate the understandability of these examples, we provide partial data dependence graphs for both Example 1.1 and Example 2.1. Each node n_c represents a cell c, containing the cells identifier as well as the list of cell references used in the formula, i.e. $(c = \rho(\ell(c)))$. Each arrow represents a data dependence between cells and the faulty references are highlighted using a red color. Note that we omitted input cells and references to input cells to keep the graph as compact as possible.

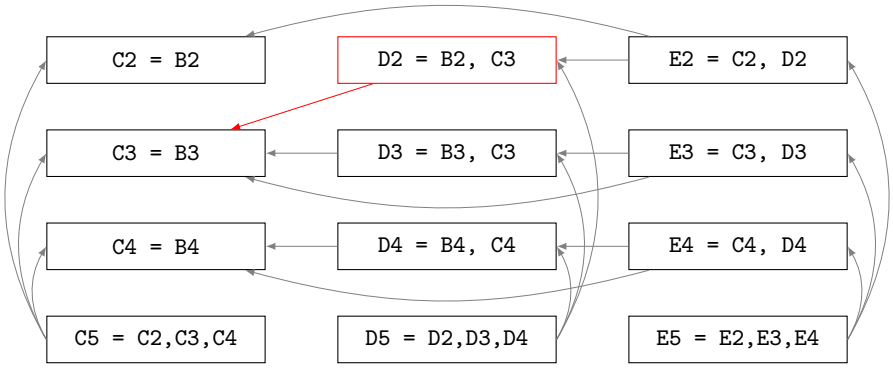


Figure 2.2: This partial program dependency graph for Example 1.1 shows the cells and their references and the data dependency arrows. The faulty cell `D2` has a red border and the faulty reference to `C3` is indicated by a red arrow.

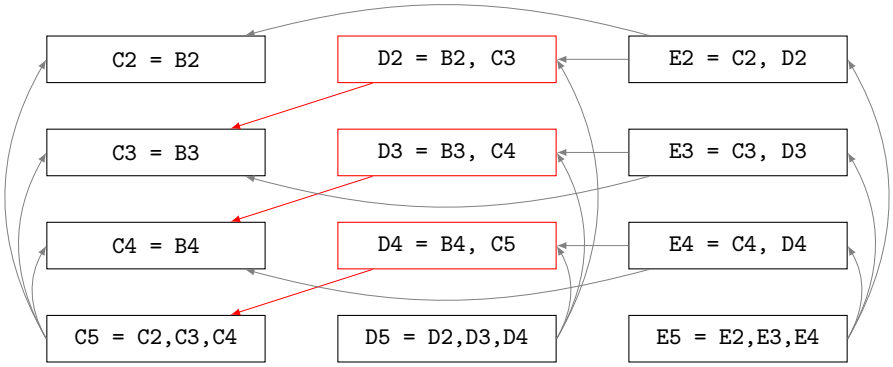


Figure 2.3: This partial program dependency graph for Example 2.1 shows the cells and their references and the data dependency arrows. The faulty references from column `D` to `C` are indicated by red arrows.

2.2 Spectrum-based Fault Localization

In previous work [15], we presented an approach to localize faults in spreadsheets based on techniques for traditional programming [31, 30, 16]. In this section, we present one such technique, spectrum-based fault localization, short SFL, in more detail, including its expected input and output as well as how this technique was evaluated.

2.2.1 Input

Fault localization techniques rely heavily on the quality and correctness of the user input. This section describes the type of information needed for SFL to work. The following paragraphs show different ways expected behavior may be indicated by using so-called testing decisions. We then examine how these testing decisions are processed and combined with the information extracted from the spreadsheet, forming the information base of the algorithm.

User Input

As the observation of erroneous behavior is the starting point of fault localization, an infrastructure is needed to allow the communication of the expected behavior. This information may take different forms, such as providing expected values, relative assertions or intervals. Any type of input the user provides regarding the value of a cell c is a testing decision $d(c) \in TD$, where TD is the set of all testing decisions. A testing decision is negative ($d(c) \in TD^-$) if it points to erroneous behavior either via X marks, expected values that differ from the computed values or violated assertions or intervals. In contrast, check marks or computed behavior inside the user-defined boundaries are positive testing decisions, the set of which we refer to as TD^+. Both TD^- and TD^+ are sets of testing decisions, so that $TD = \{TD^- \cup TD^+\}$.

For Example 1.1, we provide three testing decisions which can be seen in Figure 2.4. The sum of salaries in `C5` and the sum in `E3` are marked by the user as expected values (✓), therefore creating positive testing decisions, whereas the total sum of salaries in `E5` produces an unexpected value, indicated by the X mark (✗). While testing decisions in the form of expected values may be provided for SFL, this additional information is not considered for the algorithm and therefore Boolean decisions such as check and X marks suffice.

	A	B	C	D	E
1		Hours	Salary	Bonus	Sum
2	Jones	17	272	26	298
3	Smith	13	208	0	☑ 208
4	Rogers	20	320	40	360
5	Total		☑ 800	66	☒ 866

Figure 2.4: The testing decisions for Example 1.1 with the two positive decisions shaded in green and the negative decision shaded in red.

Algorithm Input

SFL uses the data flow structures of a spreadsheet's DDG by creating slices [28] from the given testing decisions.

Definition 2.2.1. (*Producers*) For spreadsheets, the backward slice of a cell c is the set of cells that contribute to the value of c and are therefore directly or indirectly referenced by c. We refer to this set as *producers* of c, and describe it recursively as

$$producers(c) := \bigcup_{c' \in \rho(\ell(c))} c' \cup producers(c')$$

The producers of a cell c can be found easily by tracing the path in the DDG. In Example 1.1, `D2` references `B2` and `C3` directly, and `B3` indirectly through `C3`. The set of producers for `D2` is therefore {`B2`,`C3`,`B3`}.

Definition 2.2.2. (*Consumers*) The *consumers* of c, also called the forward slice, are all cells that are directly or indirectly dependent on c, in other words use c. They are defined as follows:

$$consumers(c) := \bigcup_{\forall c' \in \text{CELLS} : c \in \rho(\ell(c'))} c' \cup consumers(c')$$

The consumers of cell `D2` are `D5` and `E2`, which directly reference `D2`, as well as `E5` using `D2` indirectly through `E2`. Slices are used for fault localization because the fault usually influences the result of each of its consumers. A cell that behaves erroneously must therefore be faulty itself or have at least one faulty cell as a producer and might in turn propagate this erroneous behavior to its own consumers. However, due to the peculiarities of spreadsheets, such as the lack of control dependencies, the term slice cannot be applied directly. As an alternative to slices, we use the function CONE, originating from hardware debugging, to compute the set of producers including the cell c.

Definition 2.2.3. (*The function* CONE) The CONE of a cell $c \in$ CELLS contains the cell c and all directly or indirectly referenced cells, therefore

$$\text{CONE}(c) = c \cup producers(c).$$

The use of CONEs shows the contribution of cells to the given testing decisions and subsequently their participation in test cases.

We previously [15] defined a test case as a tuple (I, O), where I is a set of input cells and their corresponding values and O is a set of result cells with their expected values. For this paper, we adapt this definition to show the correlation of testing decisions and test cases. Any cell with a testing decision can function as output for a test case, making expected values for output cells optional. This means that any formula cell with a testing decision may be an output for a test case, even if it is referenced by other cells, i.e. not a result cell.

Additionally, we split a single test case into multiple test cases by creating a new test case $t = (I, \tilde{o})$ for each cell $\tilde{o} \in O$. This means that every test case t has a single cell $\tilde{o}$ with a testing decision $d(\tilde{o}) \in TD$. If this decision is negative, the test case fails, i.e. $t \in \mathrm{TC_F}$, otherwise the test case passes. Assuming TC is the set of all test cases passing and failing, we therefore write

$$\mathrm{TC_F} := \left\{ t = (I, \tilde{o}) \in \mathrm{TC} \,\middle|\, d(\tilde{o}) \in TD^- \right\},$$
$$\mathrm{TC_P} := \left\{ t = (I, \tilde{o}) \in \mathrm{TC} \,\middle|\, d(\tilde{o}) \in TD^+ \right\}.$$

Test cases allow reasoning over cells that do not have testing decisions by using the Cone of a cell $\tilde{o}$ to get the set of participants for the given test case. In other words, a cell c contributes to a test case $t = (I, \tilde{o})$ if $c \in \textsc{Cone}(\tilde{o})$. For the given testing decisions for Example 1.1, we can infer two passing test cases with the output cells `C5` and `E3` as well as one failing test case, originating from `E5`. We refer to these test cases by using their output cells and their attached testing decision. For example, $\texttt{E5}_{\texttt{✗}}$ should be read as the failing test case $t = (I, \texttt{E5})$, where $d(\texttt{E5}) \in TD^-$. Given that the expected input values are equal to the values provided in the spreadsheet, the inputs need not be provided explicitly.

The contribution of cells to each test case in Example 1.1 can be seen in Table 2.1. Note that the input cells in column `B` have been excluded from the test cases, as they are assumed to be correct.

Table 2.1: Test case participation marking each cell's participation in the test cases for Example 1.1. The test cases are referred to with the output cell and the type of testing decision attached to it, i.e. $\texttt{E5}_{\texttt{✗}}$ represents the test case $t = (I, \texttt{E5})$ with $d(\texttt{E5}) \in TD^-$.

c	C2	C3	C4	C5	D2	D3	D4	D5	E2	E3	E4	E5
$\texttt{E5}_{\texttt{✗}}$	•	•	•		•	•	•		•	•	•	•
$\texttt{E3}_{\texttt{✓}}$		•				•				•		
$\texttt{C5}_{\texttt{✓}}$	•	•	•	•								

Issues with User Input

We now have two sources of information: The testing decisions given by the user, judging the values of specific cells, and the Cones for test cases, allowing reasoning over multiple cells.

A failing test case $t \in \mathrm{TC_F}$ indicates the existence of at least one faulty cell in the cells that contribute to t. The inversion of the argument, meaning no faulty cell exists which contributes to a passing test case, is not true. It is possible that the negative effects of the fault are reversed, leading to the *coincidental correctness* of a cell.

Definition 2.2.4. (*Coincidental Correctness*) A cell c which directly or indirectly references a faulty cell f and whose computed value does not differ from its expected value is coincidentally correct.

Such a behavior occurs if the fault is not actually used to compute the value, a computation nullifies the faulty value, for example by multiplying by zero, or an additional fault reverses the effect of the first. This means that more testing decisions do not necessarily lead to a better result, with recent research showing that too many positive testing decisions may influence the result negatively due to coincidental correctness [14].

Additionally, the user might make errors when placing testing decisions.

Definition 2.2.5. (*Oracle Mistake*) The oracle, often the end-user, provides information about the expected behavior. If such information is incorrect, we refer to this error as an oracle mistake.

These mistakes may lead to an erroneous or faulty cell being marked as correct or a correct cell to be marked as erroneous.

2.2.2 Algorithm

SFL uses the data from the test case participation of each cell to compute a similarity coefficient which indicates fault likelihood. To provide the needed data, SFL creates a hit-spectra matrix from the test cases, assigning each cell a column j and each test case a row i. This binary matrix maps participation of cells in test cases and is therefore also called participation matrix. Additionally, an error vector is needed describing each test case as passing or failing. Figure 2.5 shows the hit-spectra matrix for Example 1.1 as well as the error vector e.

$$
\begin{array}{c}
\\
\texttt{E5}\text{✗} \\
\texttt{C5}\text{✓} \\
\texttt{E3}\text{✓}
\end{array}
\begin{array}{c}
\begin{array}{cccccccccccc}
\texttt{C2} & \texttt{C3} & \texttt{C4} & \texttt{C5} & \texttt{D2} & \texttt{D3} & \texttt{D4} & \texttt{D5} & \texttt{E2} & \texttt{E3} & \texttt{E4} & \texttt{E5}
\end{array} \\
\begin{pmatrix}
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\end{array}
\quad
\begin{array}{c}
e \\
\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}
\end{array}
$$

Figure 2.5: The hit-spectra matrix, which maps cells to test cases, and error vector e, indicating a test case to be failing if the value is one.

This matrix allows for fast computation of the number of failing and passing test cases a cell contributes to. These numbers are then used to compute a similarity coefficient which shows the correlation between occurring errors and the cells involved in the error. This means that a high coefficient shows a high correlation and therefore fault likelihood. We use the Ochiai coefficient [4] for our implementation, which is defined as follows:

$$
\mathrm{FL}_{\mathrm{Ochiai}}(c) = \frac{|\,\mathrm{TC_F}(c)|}{\sqrt{(|\,\mathrm{TC_F}(c)| + |\,\mathrm{TC_P}(c)|) \cdot |\,\mathrm{TC_F}\,|}},
$$

where $\mathrm{TC_F}$ denotes the set of all failed test cases, $|\,\mathrm{TC_F}(c)|$ is the number of failing test cases the cell c contributes to and $|\,\mathrm{TC_P}(c)|$ is a similar definition for passing test cases. Hofer *et al.* [14] evaluated an extensive amount of similarity coefficients for spreadsheets, finding that the group containing Ochiai performed best in the worst case scenario, where the user must inspect all cells with an equal coefficient before inspecting the faulty one.

SFL assumes a single fault spreadsheet, in which case the faulty cell must be contributing to all failing test cases, barring user errors in the testing decisions. Therefore, a cell only receives the maximum fault likelihood of one if it participates in all available failing test cases and no passing ones. This is measured by the confidence factor, which sets the failing test cases for c in proportion to the total number of test cases. This also means that even few test cases allow high result values.

Additionally, the passing test cases are considered proportionate to the failing test cases in the Ochiai coefficient. If the proportions $|\,\mathrm{TC_F}(c)| : |\,\mathrm{TC_P}(c)|$ and $|\,\mathrm{TC_F}(c)| : |\,\mathrm{TC_F}\,|$ stay identical, the same result is produced regardless of the number of test cases. For example, a cell that participates in one failing and one passing test case would receive the same coefficient if it participated in ten failing and ten passing test cases, assuming $|\,\mathrm{TC_F}(c)| : |\,\mathrm{TC_F}\,|$ remains unchanged.

Table 2.2 shows the computed values for each cell in Example 1.1, where the total number of failed test cases $|\,\mathrm{TC_F}\,|$ equals one. For our example, `E2`, `E4`, `E5`, `D4` as well as the faulty cell `D2` show a fault likelihood of one, which is the maximum for the Ochiai coefficient.

Table 2.2: Fault likelihood values for Example 1.1 with SFL using the Ochiai similarity coefficient with $|\,\mathrm{TC_F}\,| = 1$. The faulty cell `D2` is shaded in gray and has a likelihood of one, along with four other cells.

Cell c	$\lvert\,\mathrm{TC_P}(c)\rvert$	$\lvert\,\mathrm{TC_F}(c)\rvert$	$\mathrm{SFL_O}(c)$
C2	1	1	0.7
C3	2	1	0.6
C4	1	1	0.7
C5	1	0	0
D2	0	1	1
D3	1	1	0.7
D4	0	1	1
D5	0	0	0
E2	0	1	1
E3	1	1	0.7
E4	0	1	1
E5	0	1	1

2.2.3 Output

SFL returns a numerical value for all cells which indicates their fault likelihood. This value is used to create a ranking. A cell ranking is a list of all cells, sorted in the order of their likelihood to contain the true fault, indicating to the user which cells to examine first. Each cell has one rank or level in the ranking and a perfect fault localization technique ranks the faulty cell at the highest level. Regardless of the position in the ranking, it may occur that one or more cells receive the same fault likelihood and

therefore share the same rank. In this case, there exists again a lack of prioritization, as all cells sharing the same rank are equally likely to be faulty. Xu *et al.* [32] referred to such occurrences as ties and described them similarly to our following definitions.

Definition 2.2.6. (*Tie*) A tie is the non-empty set of all cells that have the same fault likelihood. Let the function $\tau(c)$ return the tie for a given cell c, so that

$$\tau(c) = \left\{\forall c' \in \textsc{Cells} : \mathrm{FL}(c) = \mathrm{FL}(c')\right\}.$$

Definition 2.2.7. (*Critical Tie*) We speak of a critical tie if one or more faulty cells are contained in a tie. Formally, we state that a given tie CT is critical if

$$\exists c_f \in CT : c_f \in F,$$

where F is the set of faulty cells.

Ties cause issues of prioritization if they consist of more than one cell. For our results in Table 2.2, we have one critical tie $CT = \{\texttt{D2}, \texttt{D4}, \texttt{E2}, \texttt{E4}, \texttt{E5}\}$ as well three other ties, $T_1 = \{\texttt{C2}, \texttt{C4}, \texttt{D3}, \texttt{E3}\}$, $T_2 = \{\texttt{C3}\}$ and $T_3 = \{\texttt{C5}, \texttt{D5}\}$. T_2 shows that only cell `C3` has a unique rank, whereas CT, T_1 and T_2 have a size greater than one, meaning that the tied cells share the same rank and cannot be prioritized further.

Confidence

The Ochiai coefficient takes the total number of failed test cases into consideration: a technique to which we refer to as confidence. This allows to set the number of test cases a cell contributes to in relation to the total number of test cases: A cell that contributes to only a small fraction of the total number of failed test cases is weighted with less impact than cells that contribute to a large fraction of failed test cases. This allows the Ochiai coefficient to reach the maximum fault likelihoods even with a small number of test cases.

When using the Ochiai similarity coefficient, SFL returns values between zero and one. For the given example, the computed fault likelihoods reached the maximum, which can be seen in Figure 2.6, where the highest ranked cells have the darkest shade of orange.

	A	B	C	D	E
1		**Hours**	**Salary**	**Bonus**	**Sum**
2	Jones	17	0.7	1	1
3	Smith	13	0.6	0.7	0.7
4	Rogers	20	0.7	1	1
5	**Total**		0	0	1

Figure 2.6: Output visualization for Example 1.1 using SFL with Ochiai, with the faulty cell with a value of one shaded in the darkest orange along with four other tied cells.

2.2.4 Metrics

In this section, we discuss possible metrics to measure the success of our approach. For now, we assume that a spreadsheet contains only a single fault, such as the fault in Example 1.1. This allows us to measure the success of the technique, using the rank of the faulty cell. The first metric is to inspect the absolute rank given to a cell, which describes the number of cells that need to be inspected to reach the faulty cell.

Definition 2.2.8. (*Absolute Rank*) The absolute rank of a cell c_f is defined as

$$\textsc{AbsRank}(c_f) = |\{c \in \textsc{Cells} : \text{FL}(c) > \text{FL}(c_f)\}| + 1,$$

returning the rank of one if the faulty cell has the highest rank, therefore counting the faulty cell itself.

As spreadsheets vary greatly in size and the absolute rank may reach high values, it is useful to set the absolute rank in relation to the size of the spreadsheet, for which we use the relative rank.

Definition 2.2.9. (*Relative Rank*) The relative rank of a cell c_f is given by the function

$$\textsc{RelRank}(c_f) = \frac{\textsc{AbsRank}(c_f)}{|C_R \subseteq \textsc{Cells}|},$$

where C_R is the subset of relevant cells in a spreadsheet, for example all cells containing formulas.

So far, we have not considered the issue of critical ties, as the absolute rank counts only cells with a rank higher than that of the faulty cell. Given that the user is presented with a critical tie, it is left to chance whether the faulty cell is inspected first or last from the set of equally ranked cells. We therefore refer to three possible scenarios which result in different ranks. Assuming the user inspects the faulty cell first, we speak of a best case scenario, in which the ranks remain unchanged, i.e. $\textsc{AbsRank}_{best}(c_f) = \textsc{AbsRank}(c_f)$. If we assume a worst case scenario, the user must inspect all cells of the critical tie before reaching the faulty cell.

Definition 2.2.10. (*Worst Case*) In the worst case scenario, all cells with an equal rank are added to the absolute rank, so that the function is defined as

$$\textsc{AbsRank}_{worst}(c_f) = |\{c \in \textsc{Cells} : \text{FL}(c) \geq \text{FL}(c_f)\}|.$$

Note that the faulty cell itself is included in this set.

Additionally, we can assume an average case scenario, where half of the critical tie needs to be inspected.

Definition 2.2.11. (*Average Case*) Assuming only half of the critical tie is inspected, we define the average case absolute rank as:

$$\textsc{AbsRank}_{avg}(c_f) = \frac{\textsc{AbsRank}_{best}(c_f) + \textsc{AbsRank}_{worst}(c_f)}{2}.$$

These definitions allow us to quantify how well the fault localization technique works. For Example 1.1, we have an absolute best case rank of one, and worst case rank of five, which is also the size of the critical tie. In the average scenario, the absolute rank would equal three. For the relative ranks, we divide the absolute rank through the number of formula cells, in this case 12. This results in $\text{RelRank}_{best}(c_{\texttt{D2}}) = 0.08$, $\text{RelRank}_{worst}(c_{\texttt{D2}}) = 0.42$ and $\text{RelRank}_{avg}(c_{\texttt{D2}}) = 0.25$.

Impact and Tie-Reduction

Based on the metrics used by Xu *et al.* [32], we propose the following metrics to measure the success of our proposed improvements.

Definition 2.2.12. (*Impact*) This metric describes the difference between the relative rank *before* the applied improvement and after. We define it as

$$\textit{Impact} = \text{RelRank}(c_f)^{before} - \text{RelRank}(\tilde{c}_f)^{after},$$

where c_f and $\tilde{c}_f$ are the highest ranked faulty cells for each ranking.

We provide two separate faulty cells as the first faulty cell found by the improved approach might not be the same as the first, in case of multiple faults. We can measure the *Impact* for the best, worst and average case scenario. For our evaluation, we will use $\textit{Impact}_{avg}$, using the average scenario relative rank as this provides us with the most information. Assuming the average case scenario, we compare the performance of the strategies against pure chance, meaning it is just as likely that the user finds the fault in the first half of the tie as it is to find it in the second half. A positive average case *Impact* means that the applied strategy improves this chance, while a negative *Impact* indicates that the previous tie provides better chances to find the fault than with the applied strategy. Note that almost all proposed strategies always improve the worst case scenario, so a negative impact does not mean that the results have worsened beyond this point.

The second metric is used only for tie-breaking strategies, which focus on improving prioritization within the tie.

Definition 2.2.13. (*Tie-Reduction*) This metric indicates the reduction of the size of the critical tie in relation to the previous size of the critical tie.

$$\textit{Tie-Reduction} = 1 - \frac{|\tau(c_f)^{after}|}{|\tau(c_f)^{before}|}$$

A high *Tie-Reduction* indicates that the strategy is successful at eliminating the lack of prioritization, but provides no information on the success of the improvement over the ranking.

3 Related Work

In this section, we briefly discuss and compare existing fault localization techniques in spreadsheets and examine their strengths and weaknesses. This comparison is a summary of our previous work [11], which discusses these algorithm in more detail. Figure 3.1 shows an overview of all techniques discussed in this section.

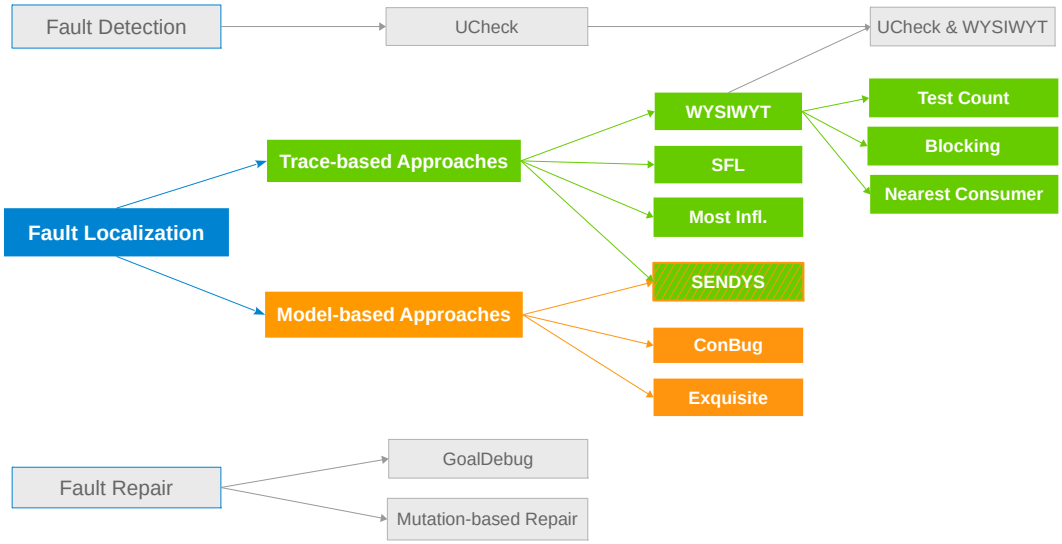


Figure 3.1: Overview spanning the discussed fault localization techniques as well as fault repair and detection techniques.

There are two basic approaches to fault localization in spreadsheets: trace- and model-based debugging. In Section 3.1, we discuss trace-based approaches, which analyze formulas and use the relationships between cells, created by cell references, to reason over fault likelihoods. The section briefly describes fault localization with WYSIWYT [27], spectrum-based fault localization [15] and an approach that searches for the most influential cell, which we refer to as MOSTINFLUENTIAL [6]. Two model-based approaches, EXQUISITE [18] and CONBUG [15], are discussed in Section 3.2. Model-based debugging uses a system description and observations to find explanations for the fault, where the description is usually formulated via constraint programming.

There also exist combined approaches, of which we discuss two: In Section 3.3.1 we examine SENDYS [15], which combines the trace-based SFL with a light-weight model-based approach. A combination of WYSIWYT fault localization with the static unit checker UCheck [2] is the focus of Section 3.3.2. While the main scope of this work is fault localization, we also briefly summarize two repair approaches, GoalDebug [1] and

mutation-based repair [13], which aim to correct the fault instead of merely locating it, in Section 3.3.3.

Note that in this work, we focus on improving SFL, which we explain in detail in Section 2 along with common definitions and terminology. However, as all of the discussed techniques have unique strengths and weaknesses, this comparison reveals additional challenges and opportunities when working with fault localization in spreadsheets.

3.1 Trace-based Approaches

3.1.1 WYSIWYT

WYSIWYT stands for 'What You See Is What You Test' and is a testing methodology for spreadsheets. Ruthruff *et al.* [27] extended this tool to include three fault localization techniques called Test Count, Blocking and Nearest Consumer.

Test Count

Test Count creates a fault likelihood value for each cell by counting the number of failing and passing test cases a cell participates in and setting these values in relation to each other. As user input, this technique requires testing decisions such as check or X marks to mark a cell's value as correct or incorrect. Slicing is used to get the producers of these cells, resulting in test cases. To compute the likelihood for each cell, the number of passing test cases is subtracted from the doubled number of failing test cases, which is therefore given more weight. The total number of test cases or failed test cases is not taken into account and the resulting value is unbounded. To limit the possible likelihood values, the resulting value is mapped to a likelihood level between zero and five, where zero indicates none and five a very high likelihood to contain the fault. This creates large ties, where cells that have different values but map to the same level cannot be prioritized any longer. One advantage of this method is that due to the small weight given to passing test cases, it is relatively robust against user errors or coincidental correctness.

Blocking

The Blocking technique works in a similar fashion to Test Count, but introduces the ability to block test cases. Blocking is based on dicing, which would exclude all those cells from the result that participate in at least one passing test case, regardless of the number of failing test cases it participates in. This would be problematic due to issues such as wrong user input and coincidental correctness. In contrast, Blocking allows a check mark in cell c to block the effect of an opposing X mark in a consumer cell of c and vice versa. For the computation of the likelihoods, only reachable or unblocked test cases are counted. However, cells are not completely removed from the result if they only participate in blocked failed test cases but rather given a very low likelihood. This enables more cells to receive a low likelihood in comparison to Test Count, even for a

small number of test cases. The disadvantage of this method is that it relies heavily on correct user input as check marks are given much more weight. This also means that it is not robust against coincidental correctness, as passing test cases are important to the blocking process. In this thesis, we try to combine the results produced by SFL with the Blocking technique as an improvement strategy.

Nearest Consumer

The Nearest Consumer technique aims to save memory and computation time by only taking direct consumers into account rather than entire slices or test cases. To compute the fault likelihoods, Nearest Consumer averages the likelihoods of all direct consumers and increases this value by one if the number of X marks is greater than the number of check marks in the direct consumers. This algorithm therefore needs initial fault likelihoods, which are set to zero for all cells except cells with X marks, which are set to a medium likelihood. Additionally, the Blocking technique is emulated for cells with a check mark and an average fault likelihood greater than zero, which receive a fault likelihood of one or very low. The initial fault likelihood is problematic, as cells without testing decisions are assumed to be correct and cells with check marks only contribute to the result via blocking. As Blocking is emulated, this technique also assumes correct testing decisions and is therefore susceptible to oracle mistakes and coincidental correctness. Due to the medium initial fault likelihood for cells with X marks, this technique can produce higher likelihoods with few test cases compared to the first two approaches, Test Count and Blocking.

3.1.2 Spectrum-based Fault Localization (SFL)

SFL is discussed in detail in Section 2.2, we will only touch on its related work here. As opposed to Test Count, SFL uses a similarity coefficient to compute fault likelihoods and takes the total number of test cases into account, allowing high confidence even with a small number of test cases. Hofer *et al.* [15] evaluated this approach using the Ochiai coefficient and compared it both to more primitive approaches, the union and intersection of CONEs of erroneous cells, and two sophisticated approaches, SENDYS and CONBUG, which are discussed in Section 3.2.1 and Section 3.3.1 respectively. SFL showed improvement over the primitive approaches with only slight increase in runtime, making it the base algorithm for the improvements discussed in this paper. Hofer *et al.* [14] compared and evaluated a wide range of similarity coefficients for SFL, finding Ochiai to be in the group performing best for worst case scenarios, meaning that all cells in a critical tie need to be inspected before reaching the faulty cell.

3.1.3 MostInfluential

MOSTINFLUENTIAL is a technique which returns the most influential cell for a given cell c, assuming it has the highest fault likelihood. As opposed to the other trace-based approaches, this algorithm only returns a single cell from the set of producers of c. As user input, the user must provide expected intervals for cells which are in turn propagated

in a separate bounding spreadsheet. These values are then checked against the actual computed values of the spreadsheet, resulting in violated intervals and therefore negative testing decisions. Instead of considering slices, this algorithm only uses the information from direct producers and consumers starting from the given cell c. It checks which of the direct producers of c has the most erroneous consumers, i.e. contributes to cells that violate an interval, and then selects this cell as a new starting point c. One disadvantage of the proposed technique is that only cells that violate an interval may be returned as most influential, which necessitates that an interval is provided for the faulty cell to return it as potentially most influential. It is also not necessarily the case that the most influential cell is faulty.

3.1.4 Strengths and Weaknesses

Trace-based approaches in general offer intuitive output in the form of a cell ranking while requiring only low complexity input in the form of X and check marks or intervals as well as short computation time. They often function better when providing several test cases, yet too many test cases may cause problems when considering coincidental correctness, user faults or multiple fault complexities. Due to the form of the output as well as the computation of the likelihoods, trace-based approaches have difficulties handling and displaying multiple faults in a spreadsheet.

3.2 Model-based Approaches

Model-based approaches translate the spreadsheet into a model and the given user input serves as the description defining the expected behavior. If the model and the description are inconsistent, we speak of a conflict, where a conflict set is the minimal set of cells required to produce a conflict. As output, model-based debugging returns a set of diagnoses, where a single diagnosis is a set of one or more cells that explain the fault. This means that model-based approaches are well equipped to handle and return multiple faults.

3.2.1 ConBug

As user input, ConBug [15] requires expected values for cells, offering a system description without the need to know the actual results of the model. Test cases are formed from these expected values, where a failed test case has an expected value that differs from the computed value. Slicing is used to create the conflict sets, using the Cone of each cell in the spreadsheet that differs from the model description, i.e. has a failing test case. These conflict sets allow the reduction of the problem size to only the conflicting cells. The model as well as the provided test cases are translated into a constraint satisfaction problem, where an *abnormal* predicate AB is added to each formula cell that might be faulty. This Boolean predicate allows single components or cells in the constraint model to fail by setting the AB value of a cell to true, meaning the formula of the cell no longer has to hold. The solver finds possible values for the AB predicates while searching for

consistent models. One possible solution, i.e. one consistent model, equals a diagnosis, consisting of all cells whose AB predicates were set to true. It is also possible to use SMT solvers instead of constraint solvers, which speeds up runtime and supports Real numbers.

3.2.2 Exquisite

Exquisite [18] also requires at least one failing test case with expected values to function properly. Additionally, assertions and multiple input values are possible user inputs, which are added to the solver as additional constraints. The conflict sets are given from the QuickXPlain algorithm and the hitting sets of these conflict sets are possible diagnoses. A hitting set H represents each conflict with at least one cell, so that the intersection of each conflict with the hitting set is not empty. The cells in a possible diagnosis (i.e. minimal hitting set) are then removed from the constraint model and the model is checked for consistency. If the model can be solved, a diagnosis is found, otherwise the diagnosis is discarded. This tool has the advantage of allowing additional constraints via the user input, therefore allowing more complex user input for expert users. Due to the use of hitting sets, the cardinality of the diagnoses is limited to the number of conflict sets found. As the constraint solver Choco is used as a back end, Exquisite is also limited to Integer values only.

3.2.3 Strengths and Weaknesses

Model-based debugging offers explanations rather than likelihoods and can therefore significantly reduce the search space opposed to trace-based approaches. Only a small number of testing decisions are required to work and additional assertions can be added to the constraint solver easily. However, the initial input required from the user is more complex, as the expected values for an erroneous cell must be provided for the algorithm to function correctly. While SMT solvers aim to improve runtimes, any solver call naturally increases the runtime over trace-based approaches. While model-based debugging supports multiple faults well, the presentation of diagnoses to the user is difficult as the diagnoses are not prioritized.

3.3 Other Approaches

3.3.1 SENDYS

Spectrum Enhanced Dynamic Slicing or Sendys [15] combines a lightweight model-based approach with the results from Sfl. The required user input shows little complexity, requiring check and X marks similar to trace-based approaches. Sendys uses the Cones of failing test cases as conflict sets. Similarly to Exquisite, the minimal hitting sets of these conflicts create diagnoses which offer explanations for multiple faults. The results from Sfl are used as a priori probabilities for each cell, so that multiplying the probabilities of each cell in a diagnosis results in a ranking of diagnoses. Finally,

these diagnosis likelihoods can be mapped back to the individual cells, allowing similar output to SFL with ranked cells. While SENDYS has a higher runtime than pure trace-based approaches, it is significantly faster than the model-based approaches. Naturally, SENDYS returns more diagnoses than the model-based approaches, which use solver calls to check the consistency of the model. It also can not entirely eliminate the issues that multiple faults pose to trace-based approaches.

3.3.2 WYSIWYT and UCheck

This approach combines the trace-based techniques from WYSIWYT with the static fault detection tool UCheck. UCheck introduces type checking to spreadsheets by inferring the type of a cell from the labels given in the tables. It warns the user if these types do not align, therefore providing helpful indicators to end-users. The combination of this tool with the WYSIWYT fault localization tools discussed in Section 3.1.1 was evaluated by Lawrance *et al.* [21] using different combination techniques. The evaluation distinguished between raw data mapping, where the computed fault likelihood levels were used, and threshold mapping, allowing only fault likelihoods greater than a certain threshold to influence the final result. To combine any two fault likelihoods, it is possible to distinguish between using the minimum, average or maximum of the two values. All three of these options were evaluated, finding that maximum had the highest effectiveness while minimum could be useful in more conservative environments.

3.3.3 Repair Approaches

As repair approaches require the same user input and face similar challenges presenting and ranking their output, we will briefly discuss two repair approaches.

Abraham and Erwig [1] developed a tool called GoalDebug, which requires assertions on cells and computes a ranked list of repair suggestions. This is achieved by converting the spreadsheet and assertions to constraints and finding solvable repair solutions. The ranking is based on heuristics, ranking repairs that change less of the original spreadsheet higher than others. The evaluation of the tool showed that the approach is successful only for a distinct type of fault, while others cannot not be repaired at all.

A second repair approach [13] is based on genetic programming, requiring expected values for erroneous cells as well as check marks for correct cells. Mutants are created based on the input, and the mutated spreadsheet is tested against the original. If the number of failed test cases is decreased, the spreadsheet has been partially repaired, and it can be mutated further. Once the number of failed test cases is zero, a possible repair has been found.

4 Issues with existing Fault Localization Techniques

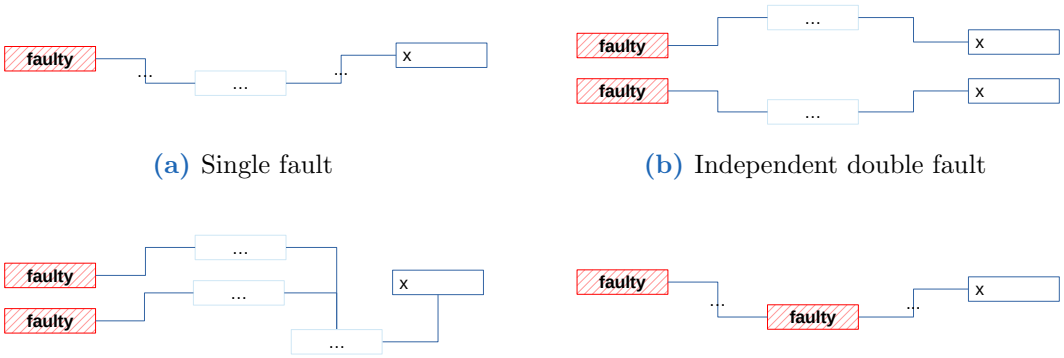
To motivate the need for improvement, in this chapter we discuss some of the challenges of fault localization in spreadsheets in more detail. Handling multiple faults in spreadsheets and the challenge it poses to trace-based approaches is discussed in Section 4.1. Issues with the faulty cell receiving a lower rank than non-faulty cells is the focus of Section 4.2. The two goals of fault localization, which we have previously identified as reduction and prioritization of the search space, should naturally be improved. We discuss limitations regarding these goals in Section 4.3.

4.1 Multiple Fault Complexity

Trace-based approaches experience great difficulty with handling multiple faults. Most therefore assume single fault complexity, as the debugging process is seen as iterative and can be repeated once a fault is fixed. However, multiple faults in spreadsheets are still challenging to trace-based approaches, causing additional noise and possibly wrong rankings. Additionally, the form of the output for trace-base approaches does not support multiple faults well, as SFL returns a ranking which prioritizes cells with higher fault likelihood. While two or more cells may have an equal fault likelihood, this is usually interpreted as "either cell `A1` or cell `B2` is faulty", whereas multiple fault feedback should be read as "both `A1` and `B2` are faulty".

Figure 4.1 provides a schematic visualization of single and double faults and how they are indicated via negative testing decisions.

If a spreadsheet contains only one fault, we speak of a single fault. Single faults are the least complex type of faults, as there is no interference and each failing test case can be correlated to the same fault. The visual representation of single faults can be seen in Figure 4.1a. If more than one fault is contained in a spreadsheet, the debugging problem has a multiple fault complexity. Multiple faults are either independent or dependent on each other, which can be seen in Figure 4.1b and Figure 4.1c respectively. Figure 4.1d shows a subset of dependent faults that are nested. The following sections describe the different types of faults closer and discuss how SFL handles these faults.



(a) Single fault

(b) Independent double fault

(c) Dependent double fault

(d) Nested double fault

Figure 4.1: Differences between single and multiple faults, where cells with red hatching are faulty, dots indicate a placeholder for an arbitrary number of intermediate cells and cells containing an X mark negative testing decisions. The path from the faulty cell to the testing decisions shows which test case the faulty cell contributes to.

4.1.1 Independent Faults

If two faults do not share any consumers prior to reaching separate negative testing decisions, we speak of two independent faults. Given that there exists at least one failing test case for each fault, the faults do not interfere with each other. The existence of a subset of these faults could be detected by checking if the intersection of two test cases is empty.

For Example 2.1, which has three faults, the faults are independent if separate testing decisions are provided before their paths meet in `E5`. Figure 4.2 shows how well-placed testing decisions in `E2` and `E4` produce two test cases, each revealing one fault indicated by the orange and blue shading. Note that the fault in `D3` is masked by coincidental correctness and therefore cannot be recognized.

	A	B	C	D	E
1		Hours	Salary	Bonus	Sum
2	Jones	17	272	26	✗ 298
3	Smith	13	208	0	208
4	Rogers	20	320	100	✗ 420
5	Total		800	126	926

Figure 4.2: Testing decisions for Example 2.1, showing two independent faults. Test case participation is indicated by orange and blue shading as well as orange and blue hatching for cells that participate in both test cases. While the faulty cells participate in only one test case each, the test cases cannot be separated due to `C2` and `C3` participating in both test cases.

To determine which cells participate in a test case, we can simply follow the dependencies for the cells `E2` and `E4` to obtain the Cone. Figure 4.3 shows the partial PDG for Example 1.1 with the dependencies from `E2` in orange and from `E4` in blue.

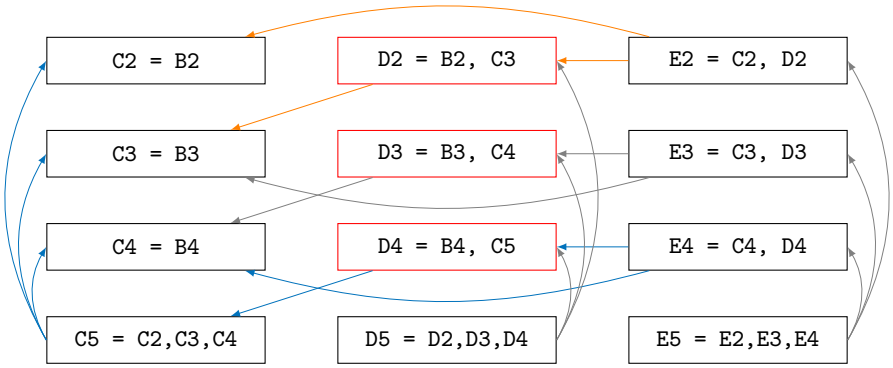


Figure 4.3: This partial program dependency graph for Example 2.1 shows the data dependencies in gray arrows, with the dependencies for each test case indicated in blue and orange. The input cells and their references are omitted to keep the graph compact.

While each fault is revealed by a separate test case, the cells `C2` and `C3` contribute to both test cases as `D4` erroneously references `C5`, which in turn consumes `C2:C4`. While the two faults are still independent, they could not be detected as such, as the intersection of the test cases is not empty. SFL would rank cells `C2` and `C3` higher than the other cells, as they participate in the most failing test cases.

As SFL takes confidence into account by considering the number of failed test cases, independent faults decrease its effectiveness. As previously stated, a cell can only receive a likelihood of one if it participates in all available failing test cases and no passing test cases. If a cell contributes to only half of the available failing test cases and no passing test cases, the computed value would be around 0.7. Therefore, multiple independent faults would decrease the confidence of the result, allowing non-faulty cells to be ranked higher than the faulty cells. This can also be observed for our example, as `C2` and `C3` receive the highest likelihood of one, while the remaining producers are ranked with a likelihood of 0.7.

4.1.2 Dependent Faults

If two or more faults participate in the same test case, i.e. share consumers, they are dependent and may influence one another. The advantage of dependent faults is that they can produce a fault localization ranking without needing separate test cases, therefore requiring less user knowledge and input.

Whether faults are seen as independent or not is defined entirely by the position and number of the testing decisions. If we provide only one testing decision for the output cell `E5` from Example 2.1, we make the faults in `D2:D4` dependent on this test case. Figure 4.4 shows this test case, where all eleven participating cells are shaded in light red. Note that this test case is significantly larger than the two separate test cases in Figure 4.2, encompassing almost all cells.

A subset of dependent faults are nested faults, where one fault f_1 is a producer of a second fault f_2, which means that any test case that applies to f_2 also applies to f_1.

	A	B	C	D	E
1		Hours	Salary	Bonus	Sum
2	Jones	17	272	26	298
3	Smith	13	208	0	208
4	Rogers	20	320	100	420
5	Total		800	126	✗ 926

Figure 4.4: Single testing decision for Example 2.1, making all three faults dependent on E5. All cells that participate in the test case are indicated by red shading.

Whichever fault contributes to more failing or less passing test cases has a higher fault likelihood. If two faults participate in the same amount of failing and passing test cases, they have the same fault likelihood. Additional testing decisions that only apply to one of the faults help to reduce the search space and let one fault rank higher than another. For nested faults, these additional testing decision can only be applied to f_1 as the second fault shares all test cases with the first. Which fault ranks higher is therefore entirely dependent on the testing decisions applied to f_1.

4.1.3 Possible Improvements

DiGiuseppe and Jones [9] examined fault interaction in more detail for several programs written in C. They also distinguished between independent faults, fault synergy and obfuscation. The independent faults are only pass/fail independent, meaning that, while the faults might influence one another, they do not change the outcome of the test cases. Fault synergy is used to describe faults that create erroneous behavior that would not have occurred for each of the single faults. Finally, fault obfuscation is defined similarly to coincidental correctness, where several faults cancel out each others negative effects, therefore hiding one another. These types of fault interaction are also applicable for the spreadsheet paradigm and knowledge of them facilitates the creation of techniques that take multiple faults into account. One such technique, called fault clustering, is used for procedural programs [25], grouping unusual executions together, assuming they are tied to the same failures. Högerle *et al.* [17] discuss further improvements on fault clustering and parallel debugging, proposing to separate the test case participation matrix into clusters with the least overlap to aid identification of multiple faults.

To the best of our knowledge, these techniques are not currently used for spreadsheet debugging, but could potentially greatly increase fault localization effectiveness. The implementation of these improvements is out of the scope of this work, yet we hope to provide a possible starting point for future work with this brief discussion.

4.2 Coincidental Correctness and Oracle Mistakes

Multiple faults are not the only way a faulty cell might receive a lower rank than non-faulty cells. Another challenge that all fault localization techniques face is that of coincidental correctness, where a cell c fails to display a symptom of the fault, even though

c is a consumer of the faulty cell. This means that the fault does not influence the value of c, $\nu(c)$. If such a cell is given a positive testing decision, the faulty cell can participate in passing test cases. This, in turn, may lead to non-faulty cells ranking higher than faulty cells, given that they participate in the same number of failing test cases and less passing test cases.

As previously mentioned, recent research [14] showed that too many positive testing decisions may adversely affect the result of the fault localization, which has been attributed to the effects of coincidental correctness. Wong *et al.* [29] also indicate that the effectiveness of positive testing decisions decrease with the number of TD^+ provided. This means that the contribution of the first TD^+ to debugging is greater than or equal to the second, the second is greater than or equal to the third and so on. To mitigate the negative effects of coincidental correctness we could therefore give less weight to positive test cases or prioritize certain test cases. There also exists clustering-based analysis to decrease the effects of coincidental correctness, arguing that a passing test case in a cluster with a failing test case has a higher likelihood of exhibiting coincidental correctness [22].

Naturally, the fault localization result is also falsified if the user sets wrong testing decisions, i.e. oracle mistakes. In this case it may also happen that the faulty cell receives a lower fault likelihood than non-faulty cells. As any debugging technique depends on correct user input, we do not focus on decreasing the risk of oracle mistakes. Please note that the user may be able to fix these faults by careful examination, whereas coincidental correctness can not be fixed in such a manner.

While we do not focus on mitigating coincidental correctness explicitly, some of the presented improvements in this work also decrease the likelihood for coincidental correctness to influence the result negatively.

4.3 Large Ties

Trace-based approaches have difficulties to reduce the search space, as they are based on heuristics. Any cell that contributes to a failing test case might contain the fault and should therefore not be excluded from the result. Ruthruff *et al.* [27] state that end-users might get frustrated if they are presented with fault localization results that do not contain the faulty cell at all, lacking familiarity with such tools. While the size of the search space is only relevant if the prioritization is insufficient, it may be discouraging for users to receive a large result set even with prioritization. The union of all cells that participate in a failing test case is an indicator of the global search space, showing how many cells would maximally need to be examined if the algorithm returned no ranking.

Model-based approaches often succeed in reducing the search space, yet their output offers little or no prioritization. All model-based approaches return diagnoses, where a single diagnosis is a set of cells that explain the fault. The global search space of model-based approaches is the union of all diagnoses, indicating the maximum number of cells that need to be inspected.

On a more local scale within the ranked cells returned from trace-based approaches, we have cells that are tied and cannot be prioritized further. As WYSIWYT approaches group their likelihoods to five levels, information is lost and up to five potentially large ties are created. SFL is not limited to five levels, instead returning likelihoods between zero and one. Ties are still likely to occur, for example if two cells participate in the same number of passing and failing test cases.

The diagnoses returned by model-based approaches are often only ranked by cardinality, ranking diagnoses with a small number of cells higher than others. We assume the output consists only of minimal diagnoses, where any super set of a diagnosis is removed from the pool of possible diagnoses. We can therefore state that two or more minimal diagnoses of the same cardinality form a tie, i.e. $d_1, d_2 \in D$ are tied if $|d_1| = |d_2|$. A tie containing the correct fault diagnosis is a critical tie, meaning a tie CT is critical if $\exists d_f \in CT : d_f \subseteq F$, where F is the set of faulty cells.

4.3.1 Reducing the Search Space

Reducing either the entire search space or the size of the critical tie can have great impact on the effectiveness of fault localization, especially when working with small cardinalities. This can be achieved either by decreasing the size of the input, for SFL the CONEs of the negative testing decisions, or by reducing the size of the output, in this case the size of the ties. In the first case, we speak of pre-processing and in the latter case of post-processing or pruning.

As a pre-processing improvement, we propose to reduce the size of the CONEs by using dynamic slicing [30], which means that only the cells that are used to compute the current value of a cell are in the CONE, as opposed to all cell references. In addition to dynamic slicing, we use grouping as both a pre- and post-processing step. Mittermeir and Clermont [23] present various functions to group cells in a spreadsheet according to their similarity. We use these grouping techniques to identify similar areas and interpret such areas as instances of the same concept. This pre-processing step allows the reduction of the CONEs by treating an identical area as a single entity. Alternatively, it is possible to use grouping to prune the resulting ties, i.e. in post-processing. If two or more cells have the same fault likelihood and belong in the same group, they can be presented to the user with the area notation, counting as only one cell in the ranking. While dynamic slicing omits cells from the search space, grouping does not reduce the number of cells but allows for a more realistic presentation, utilizing the unique properties spreadsheet programming offers.

As these techniques work on the level of cells, they can only be applied to trace-based approaches which return a cell ranking. The diagnoses returned by model-based approaches cannot be reduced by grouping or dynamic slicing.

4.3.2 Prioritization of the Search

Only trace-based approaches can apply pre- and post-processing to reduce the search space. Even if we successfully decrease the size of the critical tie, it is likely that ties remain that cannot be reduced further. Therefore, the issue of lack of prioritization persists in both trace- and model-based approaches. The solution to this problem is to break the existing ties by finding additional ways to prioritize faulty cells.

The goal of tie-breaking is to create a ranking within the tie that increases the effectiveness of the approach, i.e. ranks the faulty cells higher than the non-faulty cells within a given tie. Xu *et al.* [32] present and evaluate several strategies to break ties, often based on positions and distance of statements, for conventional, imperative programming languages. Additionally, Hermans *et al.* [12] provide an overview of metrics and code smells in spreadsheets, which can be used as heuristics, for example by ranking cells containing many operations higher than others.

As these strategies are based purely on heuristics and likelihoods, it is possible that the performance of the fault localization is diminished. While only the prioritization of the cells within the ties is changed, meaning that no cell can move from one tie to a differently ranked tie, the performance of tie-breaking is measured considering the average case scenario. While the aim of tie-breaking is to achieve a best case scenario, i.e. the faulty cell is inspected before all others in the tie, unsuccessful tie-breaking may rank other cells first. If the cell is ranked in the lower half of the critical tie, the performance of the broken tie is worse than the average case scenario. In the worst case scenario the faulty cell is ranked last within the critical tie.

5 Improvements

In the previous chapter, we presented several issues regarding fault localization. To alleviate some of these issues, we propose a number of strategies to improve the results returned by spectrum-based fault localization in this chapter. We focus on improving single-fault debugging by reducing the number of cells that need to be inspected on one hand and providing additional prioritization for the existing ranking on the other hand. We distinguish cell reduction from tie-breaking, where the former either reduces the search space or resulting ranking, thereby potentially improving even the best case fault localization results. In contrast, improvements resulting from tie-breaking are limited to creating a prioritization within the critical tie while the number of cells in the ranking remains the same.

In Section 5.1, we present the Grouping approach, which allows to cluster similar cells to a single unit. Two input reduction strategies, which improve the result by removing cells from the search space, are discussed in Section 5.2. To improve prioritization, we use tie-breaking strategies, which are discussed in Section 5.3.

5.1 Grouping

One distinct difference between spreadsheet programming and functional programming is the repeating or copying of formulas. Hermans *et al.* [12] identify duplicated formulas as a form of code smell, as code duplication is often seen as a potential for errors in traditional programming. They also state that some of their test subjects do not perceive duplication as dangerous, indicating a wide acceptance of such copying behavior that is even facilitated by such functionality as Autofill, which allows quick copying for all filled rows. Spreadsheets are therefore often row- or column-based, containing the same information and formulas for different input values. In Example 1.1, we have nearly identical formulas for all employees, and the deviation from the norm is the actual fault. The duplication of formulas is also supported in spreadsheet programs by differentiating between relative and absolute cell references. While relative cell references change if the formula of the cell is copied to different coordinates, absolute references continuously point to the same row and column as before.

Definition 5.1.1. (*The function* P) The function $\mathrm{P}(\ell(c))$ returns the set of all cell references used in the expression $\ell(c)$. References point either to a single cell c or an area of cells $c_1 : c_2$ and indicate which part of the coordinates are relative or absolute.

The function P must not be confused with ρ, which returns the set of all *cells* that are referenced by an expression e. For an area reference $c_1 : c_2$, ρ returns all cells contained

in the area, whereas P returns only the reference itself, i.e. the start and end cell and which dimensions are set as absolute.

Definition 5.1.2. (*Relative Cell Reference*) A reference $r \in \mathrm{P}(c_1)$ is relative if the location of the referenced cell $r_{c_1} \in \textsc{Cells}$ is relative to the position of c_1. This means that when the formula is moved or copied to a new cell c_2, it refers to a new cell $r_{c_2} \in \textsc{Cells}$, where

$$(\varphi_x(r_{c_1}) - \varphi_x(c_1) = \varphi_x(r_{c_2}) - \varphi_x(c_2)) \wedge$$
$$(\varphi_y(r_{c_1}) - \varphi_y(c_1) = \varphi_y(r_{c_2}) - \varphi_y(c_2)).$$

In contrast, an absolute cell reference continuously points to the same cell even if the formula is copied or moved to another cell.

Definition 5.1.3. (*Absolute Cell Reference*) An absolute cell reference $a \in \mathrm{P}(c)$ refers to a cell $a \in \textsc{Cells}$, regardless of the position of the referring cell c.

So far we used relative references in our examples, as they are easier to read and absolute references are only necessary for copying or moving formulas. To communicate when a cell reference is absolute, we precede each coordinate with a dollar sign (`$`). For example, `$A$1` will always point to the first cell in the first row of the spreadsheet. It is also possible to make only one coordinate absolute while the other remains relative by omitting the dollar sign where the reference should be relative: the reference `$A1` will continue to point to the first column, but when moving or copying the formula, the reference updates its row according to the position of the cell which holds the reference. The reference `A$1` will, in turn, always reference the first row with a variable column. For area references, it is possible to set each dimension of the start and end coordinate individually, allowing combinations such as `$A1:B$2`.

This notation to denote absolute and relative references is widely used in most spreadsheet environments and is the one most end-users are familiar with. It allows users to decide which cells are variable and which inputs should remain constant, enabling them to copy formulas many times without changing or updating them manually. We propose to use this structural information to group adjacent and copied formulas, as they can be perceived by the user as a single formula with several instances.

As a prerequisite, we need to recognize copied formulas: As the cell references are updated each time, a relative reference to `A1` from `B1` will be updated to `A2` in `B2` and so on. Analyzing formulas in this notation is difficult and requires significant computation time, as the relative positions need to be computed for each reference in a formula.

R1C1

There exists an alternative notation for cell references, commonly referred to as `R1C1`, which takes the relative positions of references to their cells into account. So far, we referenced cells with their alphanumerical representation, also called `A1` notation, where the coordinates of a cell are given by the letter of the column and the row number. The `R1C1` notation uses numerical values, so that given a cell c in column $\varphi_x(c)$ and row

$\varphi_y(c)$, the absolute reference to c can be written as $\texttt{R}\varphi_y(c)\texttt{C}\varphi_x(c)$. For relative references, instead of providing the absolute column or row, the distance from the given coordinate o is provided in brackets (`[ ]`), formally defined as

$$\texttt{R[}\varphi_y(c) - \varphi_y(o)\texttt{]C[}\varphi_x(c) - \varphi_x(o)\texttt{]},$$

where c is the referenced cell and o is the formula cell containing the reference. A relative reference from `B2` to `A1` would therefore be `R[-1]C[-1]`, and the formula copied to `B3` would remain identical, only now pointing to `A2`.

Absolute and relative references can also be mixed in this notation, stating an absolute position when omitting the brackets and a relative distance if the brackets are provided. Within the brackets, integers may be both negative or positive, pointing either to the coordinate before or after the current cell. To point to cells in the same row or column, the number is simply omitted, so that a reference to `RC[-1]` in c points to the cell in the same row but one column before this cell.

An example of this notation is provided in Figure 5.1, where the formulas of Example 1.1 are displayed in `R1C1` notation. While `A1` is easier to read, `R1C1` makes irregularities in copied formulas more apparent. This notation allows for a straight-forward comparison between two formulas, which are considered identical if their formulas are equal in `R1C1` notation.

	1	2	3	4	5
1		Hours	Salary	Bonus	Sum
2	Jones	17	=RC[-1]*16	=IF(RC[-2]>15; R[1]C[-1] /8;0)	=SUM(RC[-2]:RC[-1])
3	Smith	13	=RC[-1]*16	=IF(RC[-2]>15; RC[-1] /8;0)	=SUM(RC[-2]:RC[-1])
4	Rogers	20	=RC[-1]*16	=IF(RC[-2]>15; RC[-1] /8;0)	=SUM(RC[-2]:RC[-1])
5	Total		=SUM(R[-3]C:R[-1]C)	=SUM(R[-3]C:R[-1]C)	=SUM(R[-3]C:R[-1]C)

Figure 5.1: The formula view of Example 1.1 in `R1C1` notation, with relative positions indicated in brackets (`[ ]`) and absolute positions, starting from `R1C1`, given without brackets.

5.1.1 Grouping Function

We now present which criteria two or more cells need to meet to be able to group them. These criteria are based on previous research by Mittermeir and Clermont [23], who propose the clustering of similar formulas to identify high-level structures. The first criterion for a group is that it conforms to the area notation defined in Section 2.1. We previously defined the area $c_1 : c_2$ as the set of cells whose coordinates lie between those of the first cell c_1 and the last cell c_2. In other words, it describes a rectangular selection of the spreadsheet, so that c_1 is the top left cell and c_2 is the bottom right cell. This means that while groups may be two dimensional, they must always form a rectangle, no concave forms of groups are possible. The second criterion is that only cells with identical formulas may form a group.

Definition 5.1.4. (*Identical Formula*) The cells c_i, c_j have identical formulas if their expression in `R1C1` is identical, i.e. $\ell_{\texttt{R1C1}}(c_i) = \ell_{\texttt{R1C1}}(c_j)$.

This definition is identical to the copy equivalence of cells proposed by Mittermeir and Clermont [23] and may be loosened in the future to include other forms of equivalence. These definitions allow us to formally define a group of cells.

Definition 5.1.5. (*Group*) An area of cells $c_1 : c_2$ forms a group if it holds that

$$\forall c_i, c_j \in c_1 : c_2 \mid \ell_{\texttt{R1C1}}(c_i) = \ell_{\texttt{R1C1}}(c_j).$$

With this definition we see that subsets of groups also form groups, provided that they can be written in area notation. Typically, we are only interested in the largest possible groups. This definition allows cells to participate in more than one group. While it may be beneficial to allow such structures in the future, in our current implementation, a single cell c maps to exactly one group G, due to the iterative process of group creation. In ambiguous cases such as c having an identical formula with both the neighbor in the previous column as well as the neighbor in the row below, i.e. forming a concave polygon, the cell is grouped with the one below, as shared column coordinates are preferred.

Adhering to these grouping criteria, we identify four distinct groups in Example 1.1, as shown in Figure 5.2a. The areas `C2:C4`, `C5:E5`, `D3:D4` as well as `E2:E4` contain identical formulas and can therefore be considered as iterations for the same concept. The faulty cell `D2` does not form a group and is therefore already highlighted.

	1	2	3	4	5
1		Hours	Salary	Bonus	Sum
2	Jones	17	=RC[-1]*16	=IF(RC[-2]>15; R[1]C[-1] /8;0)	=SUM(RC[-2]:RC[-1])
3	Smith	13	=RC[-1]*16	=IF(RC[-2]>15; RC[-1] /8;0)	=SUM(RC[-2]:RC[-1])
4	Rogers	20	=RC[-1]*16	=IF(RC[-2]>15; RC[-1] /8;0)	=SUM(RC[-2]:RC[-1])
5	Total		*=SUM(R[-3]C:R[-1]C)*	*=SUM(R[-3]C:R[-1]C)*	*=SUM(R[-3]C:R[-1]C)*

(a) The formula view of Example 1.1, grouped to four groups with the faulty cell isolated.

	1	2	3	4	5
1		Hours	Salary	Bonus	Sum
2	Jones	17	=RC[-1]*16	=IF(RC[-2]>15; R[1]C[-1] /8;0)	=SUM(RC[-2]:RC[-1])
3	Smith	13	=RC[-1]*16	=IF(RC[-2]>15; R[1]C[-1] /8;0)	=SUM(RC[-2]:RC[-1])
4	Rogers	20	=RC[-1]*16	=IF(RC[-2]>15; R[1]C[-1] /8;0)	=SUM(RC[-2]:RC[-1])
5	Total		=SUM(R[-3]C:R[-1]C)	*=SUM(R[-3]C:R[-1]C)*	*=SUM(R[-3]C:R[-1]C)*

(b) The formula view of Example 2.1, with four identifiable groups, with one group comprising all three faulty cells `D2:D4`.

Figure 5.2: The cells of Example 1.1 and Example 2.1 grouped according to the given criteria.

5.1.2 Post-Processing

How can we use this grouping function to improve fault localization? One possible use of Grouping is to decrease the appearance of the critical tie produced by trace-based algorithms. By grouping cells in areas that have the same fault likelihood, the size of the tie reflects the number of unique formulas the user needs to inspect, and is therefore

a more accurate reflection of the user's search space. For post-processing, we propose an additional criterion for a group $c_1 : c_2$, so that it must hold that

$$\forall c_i, c_j \in c_1 : c_2 \mid \mathrm{FL}(c_i) = \mathrm{FL}(c_j).$$

This strategy is applied to the existing ranking of cells by their fault likelihood. Each critical tie is analyzed and whenever the grouping criteria match for two or more cells, we form a single unit represented by the area notation. The ranking then consists of the remaining single cells which could not be grouped as well as the areas for grouped cells.

The disadvantage of post-processing the results of fault localization is that the rankings are already computed and can therefore not be improved. It is possible that testing decisions or cells with slightly differing formulas break up areas that could otherwise be grouped. For Example 1.1, it is not possible to group any cells in ties, as the positive testing decision in `E3`✓ allows the cells in row 3 to be ranked lower than the others. However, if we assume a single negative testing decision in `E5`✗, Post-Process Grouping would indeed reduce the size of the critical tie.

5.1.3 Pre-Processing

Another possibility for Grouping to facilitate fault localization is to group the elements in the spreadsheet beforehand, thereby allowing more information to be passed to the fault localization algorithm. This can be achieved by collapsing a group of cells to a single entity representing the group, assuming their identical formulas point to unique instances of the same functionality. A testing decision provided for a cell c in a group G can then be applied to the entire group, which provides additional information and in some cases even enables us to remove coincidental correctness.

When using groups as basis for fault localization, we need a more complex approach for group creation. While two or more formulas may use the same syntax, they cannot always be seen as instances of the same concept, therefore differing semantically. For example, cells `C5:E5` in Figure 5.2a are a group as they form an area and their formulas are identical, yet any testing decision provided for the sum of column `C` does not apply to the other sums. While the cells execute the same action, they perform this action on different sets of data (`Salary`, `Bonus` and `Sum`). It is therefore necessary to analyze the references used by a group and find out whether information is lost in the group creation process. We call these groups collapsible or type-safe, as they do not only have identical formulas, but also execute these operations on the same type of data.

As we wish to analyze all cell references, it is necessary to group constant cells as well, which we previously omitted from the grouping process. For the purpose of type-safe groups, an area of cells can be collapsed if they are of the same type of constant, i.e. number, string, Boolean, error or blank. Their values need not be identical. Note that blank cells have a special status in spreadsheets and can often be assumed to be initialized to empty values. This means that we could allow groups of blank cells to be grouped with any other type. For now, this is ignored and a group of blanks is seen as its own type-safe group.

Collapsible Groups

Let us now inspect the steps needed to create collapsible groups with the aid of Algorithm 1, which shows the rough order of execution for the required functions. As a first step, we create groups from all formula and input cells in the spreadsheet, adhering to the grouping criteria previously defined in Section 5.1.1. As a result, each input or formula cell $c \in$ Cells is now part of a group $G \in$ Groups, where G is an area of cells of the same type. For our example, these groups are identical to the grouped cells in Figure 5.2a. As we mentioned before, each cell c belongs to exactly one group, with column-based groups treated preferentially. In the future, allowing cells to belong to more than one group may improve this kind of Grouping, as it loosens the criteria needed for groups to be collapsible.

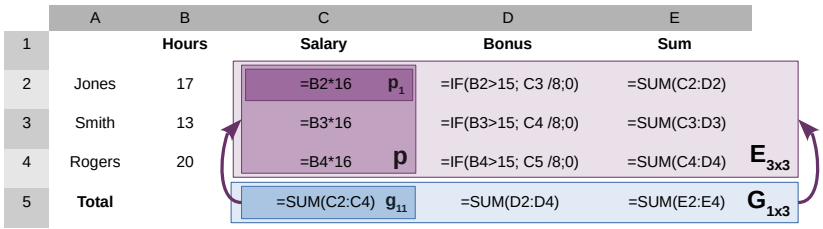
Algorithm 1 Creation of type-safe groups

Input: initial groups Groups for input and formula cells
Output: type-safe groups TypeSafeGroups

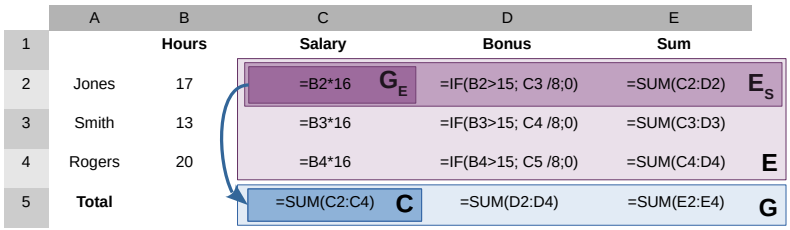
```
1:  init TypeSafeGroups
2:  for all G ∈ Groups do
3:      checkAndSplitGroup(G)
4:  end for
5:  procedure checkAndSplitGroup(G)
6:      SubGroups ← {G}
7:      for all p ∈ getReferences(G) do
8:          for all G_S ∈ SubGroups do
9:              Checked ← splitDimensions(G_S, p)
10:             CheckedSubGroups ← add(Checked)
11:         end for
12:         for all G_S ∈ CheckedSubGroups do
13:             d ← getGrowthDimension(G_S, p)
14:             E ← getEntireReferencedArea(G_S, d, p)
15:             SubAreas ← splitReferencedArea(E, G_S, d)
16:             for all E_S ∈ SubAreas do
17:                 GroupsInArea ← getGroupsInArea(E_S, TypeSafeGroups)
18:                 for all G_E ∈ groupsInArea do
19:                     SplitGroups ← splitGroupByReferencedArea(G_S, G_E, p)
20:                     NewSubGroups ← addAll(SplitGroups)
21:                 end for
22:             end for
23:         end for
24:         SubGroups ← NewSubGroups
25:     end for
26:     TypeSafeGroups ← addAll(SubGroups)
27: end procedure
```

Each group $G \in$ Groups needs to be checked for type safety and possibly split up so that each subgroup $G_S \subseteq G, G_S \in$ TypeSafeGroups. Any Group G_κ that contains only constants of the same type is checked during creation and therefore $G_\kappa \in$ Groups $\rightarrow G_\kappa \in$ TypeSafeGroups. In our example, the cells `B2:B4` form such a group of constants. Any groups representing formula cells need to be analyzed recursively, ensuring that the references are of the same type for each cell in the group. As an example for formula groups, let us analyze the group `C5:E5` that needs to be checked for type safety.

Let $G \in$ Groups be an area of dimension $r \times c$, with r as the number of rows and c as the number of columns in the group. We denote its elements as $g_{11} : g_{rc}$, where g_{11} is the first and g_{rc} is the last cell in the area. If G contains only one cell, i.e. $r \cdot c = 1$, the group is indivisible and no further checks are needed to state that $G \in$ TypeSafeGroups. Our example group `C5:E5` has the dimensions 1×3. We discuss how this group is checked and split in the next paragraphs, while also describing each function call in more detail. As a visual aid, Figure 5.3 shows a schematic overview for the given example by highlighting and labeling the relevant areas.



(a) Example 2.1 with the group G that needs type-checking highlighted in light blue and the entire area E referenced by this group highlighted in light purple. Additionally, we see the cells g_{11}, the first cell of group G, as well as the area p, referenced by g_{11}, and its first cell p_1.



(b) Example 2.1 with the group G and referenced area E highlighted in light blue and light purple respectively. We see the area E_S, which is required to be type-safe for the cells in G to be collapsible. E_S contains G_E, which is one of the type-safe groups contained in this area. We map the coordinates of G_E back to our cell g_{11}, which is now considered a type-safe group C.

Figure 5.3: An overview of the type checking process, with the entire referenced area E in purple and the area of origin G in blue.

- GETREFERENCES (line 7): This function requires a group G as input and returns the set of references used by the first cell in G. As all cells in G have identical formulas, we use the first cell, g_{11}, as a representative. For this cell, we get all references with the function P from Definition 5.1.1. A single reference $p \in \mathrm{P}(\ell_{\texttt{R1C1}}(g_{11}))$ points to a single cell or an area of cells and indicates which parts of the reference are absolute or relative. Our example group `C5:E5` has one area reference $p =$ `R[-3]C:R[-1]C`. We analyze each reference p in relation to G or all of its subgroups.

- SPLITDIMENSIONS (line 9): This function takes a group or subgroup of cell G_S and a reference p as an input. It returns a set of subgroups, for which it holds that for all $g_{ij} \in G_S$, the dimension of the area referenced by p is the same. Dimension changes can occur if p references an area and uses opposing relative or absolute references for the column, row or both. For example, `$A1:B2` sets the column in the first part of the area as absolute, but the column in the ending coordinate as relative. This means that if $c > 1$, i.e. G expands horizontally, the dimensions for p differ for each column, increasing in size. If that is the case, we conclude that not all cells in this group operate on the same set of data, as the size of the referenced area is not the same. If such dimension changes occur for one or more dimensions of the referenced area, we split G along the differing dimensions into several subgroups. For this example, we split G into c groups, each of width one, representing one column, i.e. $G \to \{G_S = g_{1s} : g_{rs}\}_{s=1}^{c}$. If opposing references occur in both dimensions, G is split into single cells and no further type checking is necessary. In our example group `C5:E5`, all references are relative and therefore this function returns the entire group `C5:E5`.

- GETGROWTHDIMENSION (line 13): As input, this function takes a subgroup G_S and a reference p. It returns the dimension $d = v \times h$ which represent by how much the reference p needs to be extended to apply to the entire group. As the cell or area represented by p is referenced only by g_{11}, we use the dimension d to compute the entire area referenced by G_S rather than just its first cell. We established that a group's height and width are given by r and c. Subsequently, we must determine which parts of the reference p are relative, as the referenced area only expands where p is relative. We define the dimension values as

$$v = \begin{cases} r - 1 & \text{if row reference is relative} \\ 0 & \text{otherwise.} \end{cases}$$

$$h = \begin{cases} c - 1 & \text{if column reference is relative} \\ 0 & \text{otherwise.} \end{cases}$$

  The previous function in line 9 ensures that the dimensions of the referenced area are identical for all cells in the group G_S. If both v and h equal 0, the referenced area is absolute regarding the group G_S, and does not need to be checked for type safety. The growth dimensions for our group `C5:E5` are 0×2, as all references are relative and the group expands horizontally by two cells, adding two cells to `C5`.

- GETENTIREREFERENCEDAREA (line 14): This function takes a group G_S, a growth dimension d and a reference p as input and returns the entire referenced area E. From the reference p, which originates from the representative cell g_{11}, we need to compute the area referenced by the entire group G_S. Given that the reference p points to an area of cells $p_1 : p_2$, we determine the entire referenced area by adding $v \times h$ to the coordinates of p_2. We define a new area $E = e_1 : e_2$, so that $e_1 = p_1$ and e_2 is at the coordinates:

$$\varphi_x(e_2) = \varphi_x(p_2) + h$$
$$\varphi_y(e_2) = \varphi_y(p_2) + v.$$

  Given the reference $p =$ `R[-3]C:R[-1]C` for our group `C5:E5`, we know that the for the first cell, the area `C2:C4` is referenced. To get the entire referenced area, we add 0×2 to `C4`, resulting in $E =$ `C2:E4` with the dimensions 3×3. See Figure 5.3a to see a schematic overview of the area E and the other cell areas required to compute it.

- SPLITREFERENCEDAREA (line 15): As input, we use the referenced area E, the subgroup G_S and the growth dimension d. This function retruns a set of areas that need to be checked for type-safety. An area referenced by a single cell reference p does not necessarily need to be type-safe, as functions such as `LOOKUP` operate on a possibly heterogeneous set of data. Additionally, there is no need for type checking if the reference is absolute in every dimension, as the use of references is comparable to constants in this case. However, if any dimension in $v \times h$ is greater than 0, we need to ensure that all cells in this dimension are collapsible. This means if we wish to collapse rows, i.e. $v = 0 \wedge h > 0$, we must ensure that each row in the referenced area is type-safe. Similarly, if we expand only vertically, i.e. $v > 0 \wedge h = 0$, we must ensure that each column is type-safe. If both v and h are greater than zero, the group expands diagonally and all cells in E must be of the same type. For our example we have the dimensions $v = 0 \wedge h = 2$, meaning we expand horizontally and all rows must be checked for type safety. We therefore return the subgroups {`C2:E2`}, {`C3:E3`} and {`C4:E4`}.

- GETGROUPSINAREA (line 17): This function takes the area of cells that needs to be checked for type-safety, E_S, and the set of all TYPESAFEGROUPS as input. It returns all type-safe groups in the area E_S. We recursively call CHECKANDSPLITGROUP for any cells that are not yet defined in TYPESAFEGROUPS, meaning we recursively analyze cell groups until a group references only input cells. Since we allow only finite spreadsheets with no circular references, we always reach such a group, provided we only analyze the relevant subsets of groups. For our example, we have $E_S =$ `C2:E2`, which is not yet in TYPESAFEGROUPS, so we call CHECKANDSPLITGROUP for E_S. In this instance, we see even without type checking that the area `C2:E2` cannot be collapsed, as these cells cannot be grouped at all: each cell in the area contains a different formula in `R1C1`. This function therefore returns only single cells for this example, i.e. {`C2`}, {`D2`}, {`E2`}.

- splitGroupByReferencedArea (line 19): This last function requires a subgroup G_S which we wish to check for type-safety, the type-safe group G_E and the reference p. We map the coordinates from G_E back to the group G_S using the relative position from G_E to p and return the resulting collapsible group C. We do this by measuring the position of G_E in E and applying the same offsets to our group G_S to create C. For our example, we can map $G_E =$ `C2` to $C =$ `C5`. See Figure 5.3b for a schematic overview of the area C and the other cell areas required to compute it. Now we know that `C5` cannot be grouped together with the other cells and add the single cell `C5` to TypeSafeGroups. The same process is done for `D2` and `E2`, resulting in the split of our given group `C5:E5` into the three type-safe subgroups {`C5`}, {`D5`}, {`E5`}.

Figure 5.4 shows our running examples with Pre-Process Grouping, taking into account the cell references used by each group. All cells that form a type-safe group share the same background color. A white background indicates that the cells could not be grouped. In Figure 5.4a, rows three and four can be merged to one representative row completely, with the faulty cell `D2` forming a separate unit. In addition to the faulty cell, the cells `E2`, `E5`, `D5` and `C5` could not be grouped. Figure 5.4b shows that we can collapse rows two and three for Example 2.1. Row four cannot be attached, as `D4` references a different type of cell, `C5`, by mistake. This also has an effect on `E4`, which can not be grouped with `E2:E3` due to its reference to `D4`.

	1	2	3	4	5
1		Hours	Salary	Bonus	Sum
2	Jones	17	=RC[-1]*16	=IF(RC[-2]>15; R[1]C[-1] /8;0)	=SUM(RC[-2]:RC[-1])
3	Smith	13	=RC[-1]*16	=IF(RC[-2]>15; RC[-1] /8;0)	=SUM(RC[-2]:RC[-1])
4	Rogers	20	=RC[-1]*16	=IF(RC[-2]>15; RC[-1] /8;0)	=SUM(RC[-2]:RC[-1])
5	Total		***=SUM(R[-3]C:R[-1]C)***	***=SUM(R[-3]C:R[-1]C)***	***=SUM(R[-3]C:R[-1]C)***

(a) The formula view of Example 1.1, with three collapsible groups, isolating row two containing the faulty cell.

	1	2	3	4	5
1		Hours	Salary	Bonus	Sum
2	Jones	17	=RC[-1]*16	=IF(RC[-2]>15; R[1]C[-1] /8;0)	=SUM(RC[-2]:RC[-1])
3	Smith	13	=RC[-1]*16	=IF(RC[-2]>15; R[1]C[-1] /8;0)	=SUM(RC[-2]:RC[-1])
4	Rogers	20	=RC[-1]*16	=IF(RC[-2]>15; R[1]C[-1] /8;0)	=SUM(RC[-2]:RC[-1])
5	Total		=SUM(R[-3]C:R[-1]C)	***=SUM(R[-3]C:R[-1]C)***	***=SUM(R[-3]C:R[-1]C)***

(b) The formula view of Example 2.1, with three collapsible groups, pointing out irregularities in row four.

Figure 5.4: An example showing type-safe cell Grouping for Example 1.1 and Example 2.1.

Applying Testing Decisions

Pre-Process Grouping affects fault localization by applying testing decisions that are submitted for a cell c in group G to the entire group G. Alternatively we can copy the decision to all cells in G, i.e. $\forall c_i, c_j \in G : d(c_i) = d(c_j)$. Fault localization techniques are provided with more cells that participate in test cases and some cases of user errors or coincidental correctness can be mitigated. This can be achieved by removing conflicting testing decisions. If two cells $c, \tilde{c} \in G$ receive conflicting testing decisions, for example $d(c) \in TD^-$ and $d(\tilde{c}) \in TD^+$, we remove the positive testing $d(\tilde{c})$ decision, assuming it is either coincidentally correct or was checked by mistake. It is also possible that $d(c)$ was marked as incorrect by accident, in which case the fault localization result is worsened as the oracle mistake is given even more weight than before. However, coincidental correctness or wrongfully set positive testing decisions are more likely to occur. This assumption is supported by a user study [27], which indicates that users are more likely to set positive rather than negative testing decisions by mistake.

It is also possible to retain only one testing decision for each group, eliminating any additional testing decisions, as they artificially increase the likelihood for cells that are used by all cells in a group. The process of copying the testing decisions leads to many additional test cases which exhibit the same behavior: Cells that are referenced by all cells in a group (for example initialization cells) participate in many test cases. It is therefore reasonable to create only a single test case for the entire group by using the union of all Cones from the group as a test case.

For our Example 1.1, using the testing decisions previously described in Section 2.2.1, we can apply the decision provided for `E3` to all elements in the group, meaning that `E4` and its Cone also participate in the test case provided by `E3`✗. This means that `E4` and `D4` rank lower than before and the size of the critical tie is reduced from five to three. Table 5.1 shows the computation of the result in more detail, with the result for the faulty cell `D2` shaded in gray.

Table 5.1: Comparison of SFL with and without Pre-Process Grouping, with the faulty cell `D2` highlighted in gray. Grouping has a positive *Impact* of 8 % over the previous ranking.

Cell c	$\text{SFL}_\text{O}(c)$	$\lvert\text{TC}_\text{P}(c)\rvert$	$\lvert\text{TC}_\text{F}(c)\rvert$	$\text{SFL}^g_\text{O}(c)$
`C2`	0.7	1	1	0.7
`C3`	0.6	2	1	0.6
`C4`	0.7	2	1	0.6
`D2`	**1**	**0**	**1**	**1**
`D3`	0.7	1	1	0.7
`D4`	1	1	1	0.7
`E2`	1	0	1	1
`E3`	0.7	1	1	0.7
`E4`	1	1	1	0.7
`E5`	1	0	1	1
$\text{AbsRank}_{avg}(c_{\texttt{D2}})$	3			2
$\text{RelRank}_{avg}(c_{\texttt{D2}})$	25 %			17 %
$Impact_{avg}$				8 %

When using Pre-Process Grouping alone, we have one critical tie $CT = \{\texttt{D2}, \texttt{E2}, \texttt{E5}\}$ as well as three lower ranked ties, $T_1 = \{\texttt{C2}, \texttt{D3}, \texttt{D4}, \texttt{E3}, \texttt{E4}\}$, $T_2 = \{\texttt{C3}, \texttt{C4}\}$ and $T_3 = \{\texttt{C5}, \texttt{D5}\}$, which has a fault likelihood of zero. Additionally, we can apply Post-Process Grouping to this result, which has no effect on the critical tie but reduces the complexity of the remaining ties. This combined Grouping approach yields $T_1 = \{\texttt{C2}, \texttt{D3:D4}, \texttt{E3:E4}\}$ and $T_2 = \{\texttt{C3:C4}\}$, reducing the size of T_1 from five to three elements, and T_2 from two to one element.

5.2 Input Reduction

Input reduction strategies improve the result by reducing the entire search space, leaving less cells to be inspected by the user. The ranking may be improved by reducing the size of the critical tie, i.e. a change in the average or worst case scenario, or even improve the best case scenario by reducing the number of cells ranked before the critical tie. This section describes two approaches that limit the input of the algorithm by reducing the size of the CONEs. Section 5.2.1 describes a technique based on dicing which removes cells based on conflicting testing decisions. Section 5.2.2 shows how Dynamic Slicing can be implemented for the spreadsheet context.

5.2.1 Blocking technique

The Blocking technique uses a modified form of the dicing debugging method [8] which assumes that any cell that contributes to a passing test case must be correct. The result of a dice is therefore the set of cells that participate in failing test cases only. As positive testing decisions are given considerable weight, this method is especially susceptible to coincidental correctness and oracle mistakes, where it is possible that the dice does not contain the faulty cell. As spreadsheet programmers are often end-users, we must assume an imperfect oracle and a lack of experience with fault localization, leading to frustration if the fault is not contained in the result at all.

As opposed to dicing, Blocking [26] does not eliminate cells that contribute to failing and passing test cases from the search space completely, instead assigning a very low fault likelihood to the affected cells. Additionally, the participation in passing test cases does not always lead to a very low ranking. In contrast to dicing, Blocking differentiates between blocked and unblocked test case participation for each cell. A testing decision $d(c)$ is blocked by an opposing testing decision, $d(\tilde{c})$, when $\tilde{c}$ is a producer of c. This means that the value of $\tilde{c}$ is used to compute the value of c. We have two scenarios where blocking can occur, (1) where an X mark blocks a passing test case and (2) where a check mark blocks a failing test case.

The first scenario, $d(\tilde{c}) \in TD^-$ blocking the testing decision $d(c) \in TD^+$, indicates coincidental correctness or oracle mistakes: Even though the erroneous cell $\tilde{c}$ produces the value in cell c, c computes an expected value. This means that either c is coincidentally correct or the user made a mistake while setting the testing decisions. In this case we remove the participation of $\tilde{c}$ and its producers in the positive test case resulting from $d(c)$, effectively giving more weight to $d(\tilde{c})$. This is beneficial if we assume that the testing decision $d(c)$ was set by mistake, i.e. $d(c)$ actually computes an unexpected value, or the value in c is coincidentally correct. However, if the negative testing decision $d(\tilde{c})$ is an oracle mistake, i.e. $\tilde{c}$ computes a correct value that was misjudged by the user, we give more weight to an incorrect testing decision. This means that the non-faulty cell $\tilde{c}$ and its producers rank higher than before and may therefore worsen the rank of the faulty cell.

The second scenario, $d(\tilde{c}) \in TD^+$ blocking $d(c) \in TD^-$, actually reduces the input size for the fault localization algorithm: If we assume correct testing decisions, we can

state that $\tilde{c}$ and its producers cannot cause the erroneous behavior in c, as $\tilde{c}$ behaves correctly. Even if $\tilde{c}$ were coincidentally correct, the erroneous value in c must be caused by a different cell and we block $\tilde{c}$ and its producers from the failing test case. Please note that if $d(\tilde{c})$ is an oracle mistake, the fault localization performance is severely impeded as potential fault candidates are ranked very low.

To implement these scenarios, we modify our previous definition of Cones to include the blocking behavior. We distinguish between positive and negative ReachableCones, where a positive ReachableCone$^+$ is blocked by any negative testing decisions and vice versa.

Definition 5.2.1. (*The function* ReachableCone) ReachableCone is a subset of a Cone and can be defined recursively as

$$\textsc{ReachableCone}^+(c) := c \cup \bigcup_{\tilde{c} \in \rho(\ell(c)) \,:\, \tilde{c} \notin TD^-} \textsc{ReachableCone}^+(\tilde{c})$$

and

$$\textsc{ReachableCone}^-(c) := c \cup \bigcup_{\tilde{c} \in \rho(\ell(c)) \,:\, \tilde{c} \notin TD^+} \textsc{ReachableCone}^-(\tilde{c}).$$

Let us consider how Blocking can be applied to Example 1.1, using the same testing decisions as seen in Figure 2.4. Cell `E5` has been marked as incorrect, meaning that all its producers, including `E3`, could contain the true fault. However, since the user has explicitly marked the value of `E3` as correct, it is unlikely that the fault is contained in the producers of `E3` or the cell itself, unless the fault is masked or the user has mistakenly marked the cell as correct. For the cells `E3` and its producers `D3` and `C3`, the participation in test case `E5`$_✗$ is therefore blocked by `E3`$_✓$.

As opposed to dicing, Blocking allows for more refinement regarding testing decisions. Cells may still participate in positive and negative test cases if the cells with conflicting testing decisions are independent, i.e. neither is a producer of the other. For Example 1.1, the cells `C2` and `C4` participate in both passing and failing test cases, as `C5`$_✓$ is independent of `E5`$_✗$. Table 5.2 shows the participation of the cells in each test case, with the blocked testing decisions indicated with an X mark.

Table 5.2: Blocked test case participation marking each cell's participation in the test cases for Example 1.1. Blocked participation is indicated with ✗.

c	C2	C3	C4	C5	D2	D3	D4	D5	E2	E3	E4	E5
E5$_✗$	•	✗	•		•	✗	•		•	✗	•	•
E3$_✓$		•				•				•		
C5$_✓$	•	•	•	•								

As previously mentioned, any cell that contributes to a failing test case, blocked or unblocked, should not be removed from the result set completely. If a cell participates in failing test cases but is not represented in the ranking due to Blocking, its fault likelihood

value is set to 1/10 of the smallest coefficient value. Table 5.3 shows the number of passing and failing unblocked test cases ($\mathrm{TC}_{\mathrm{P,U}}(c)$ and $\mathrm{TC}_{\mathrm{F,U}}(c)$ respectively) and the computed fault likelihood for each cell in Example 1.1. While the rank for the faulty cell D2 could not be improved, the ties $T_1 = \{$C2, C4, D3, E3$\}$ and $T_2 = \{$C3$\}$ could be changed to $T_1' = \{$C2, C4$\}$ and $T_2' = \{$C3, D3, E3$\}$, improving prioritization.

Table 5.3: Fault likelihood values for SFL with and without Blocking and the resulting ranks. The faulty cell D2, shaded in gray, has the highest likelihood along with four other cells. While the rank of the faulty cell remains unchanged, Blocking improves the prioritization for the non-faulty cells.

Cell c	$\mathrm{SFL}_{\mathrm{O}}(c)$	$\lvert \mathrm{TC}_{\mathrm{P,U}}(c) \rvert$	$\lvert \mathrm{TC}_{\mathrm{F,U}}(c) \rvert$	$\mathrm{SFL}^{b}_{\mathrm{O}}(c)$
C2	0.7	1	1	0.7
C3	0.6	2	0	0.07
C4	0.7	1	1	0.7
C5	0	1	0	0
D2	**1**	**0**	**1**	**1**
D3	0.7	1	0	0.07
D4	1	0	1	1
E2	1	0	1	1
E3	0.7	1	0	0.07
E4	1	0	1	1
E5	1	0	1	1
$\mathrm{AbsRank}_{avg}(c_{\mathtt{D2}})$	2.5			2.5
$\mathrm{RelRank}_{avg}(c_{\mathtt{D2}})$	21 %			21 %
$Impact_{avg}$				0 %

5.2.2 Dynamic Slicing

One possibility to reduce the number of cells that require manual inspection is to limit which cells are even in the slice or CONE. There exist several improvements on the original slicing method [28] which reduce the size of the slice further. One of those improvements is Dynamic Slicing, and in this section, we discuss how this technique can be applied to spreadsheets. Dynamic Slicing, as opposed to the static slicing used in SFL so far, follows only those references actually used to evaluate the value of a cell. Given an IF statement or any other conditional function such as VLOOKUP or HLOOKUP, dynamic references are only those references that are used to compute the value of the cell based on the evaluation of the condition statement.

Definition 5.2.2. (*The function* $\widehat{\rho}$) This function $\widehat{\rho}(e)$ returns the set of cells that are used to evaluate the expression $e \in \mathcal{L}$.

It is equal to the function ρ for all non-conditional expressions e, where it holds that $\widehat{\rho}(e) = \rho(e)$. If e is a conditional expression, for example of the form IF(e_1;e_2;e_3), then

$$\widehat{\rho}(e) = \widehat{\rho}(e_1) \cup \begin{cases} \widehat{\rho}(e_2) & \text{if } [\![e_1]\!] = \mathit{true} \\ \widehat{\rho}(e_3) & \text{if } [\![e_1]\!] = \mathit{false} \\ \varnothing & \text{otherwise.} \end{cases}$$

This function allows us to redefine Cones to include only those references which are actually used for the evaluation of the cell.

Definition 5.2.3. (*The function* DynCone) DynCone is a subset of a Cone and can be defined recursively as

$$\textsc{DynCone}(c) := c \cup \bigcup_{\tilde{c} \in \widehat{\rho}(\ell(c))} \textsc{DynCone}(\tilde{c}).$$

Let us consider Example 2.1, where the faulty cell `D3` references `C4` by mistake, yet this faulty reference is only evaluated if the condition is met. The function $\rho(\texttt{D3})$ returns the cells `B3` and `C4` as references, resulting in $\textsc{Cone}(\texttt{D3}) = \{\texttt{D3,C4,B4,B3}\}$. If we use the dynamic variants of these formulas, we get $\widehat{\rho}(\texttt{D3}) = \{\texttt{B3}\}$ and $\textsc{DynCone}(\texttt{D3}) = \{\texttt{D3,B3}\}$, as `C4` is never actually used in the evaluation of `D3`.

Dynamic Slicing may eliminate those cases of coincidental correctness where a faulty value was not propagated further due to a condition not being met. For our example, any testing decisions applied to `D3` or its consumers will not reach cells `C4` and `B4`, as they are not used in the computation of `D3`. Please note that in this case, `D3` itself is faulty, and therefore still affected by the coincidentally correct testing decision.

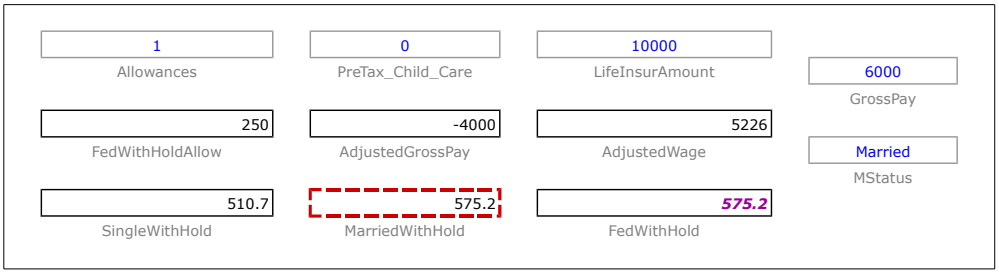
The use of DynCones potentially leads to a large reduction of Cones and therefore the search space, as entire branches of spreadsheets need not be investigated. However, if a spreadsheet uses only few conditionals or there are no cell references in the branches, this improvement might not show any effect at all.

One disadvantage of Dynamic Slicing is that all conditions in the relevant formula cells need to be evaluated to be able to compute the DynCones. We can therefore expect a significant increase in runtime, especially for spreadsheets with large amounts of conditionals.

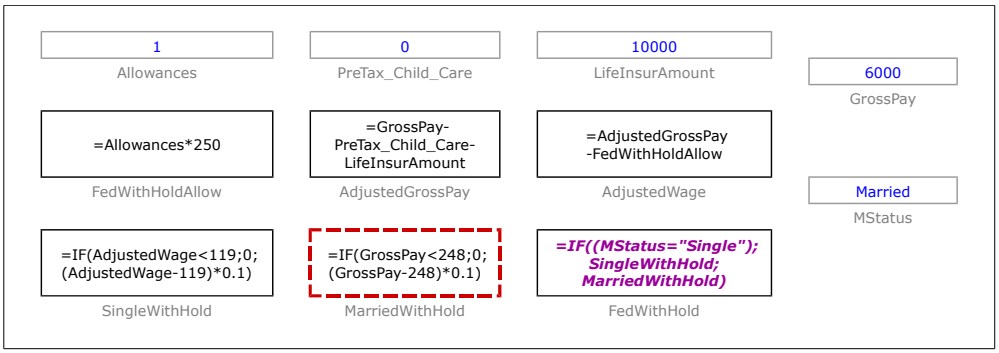
Example

While our previous examples use conditional statements, due to the structure of the spreadsheet the resulting fault likelihoods remain identical regardless of the use of Dynamic Slicing. Let us therefore introduce the additional Example 5.5, where Dynamic Slicing has a noticeable effect on the fault localization result.

This example shows a modified version of the `Payroll` tasks from the Burnett corpus [27]. As these spreadsheets were implemented in Forms/3, an alternative spreadsheet language to our previous examples, we use cell labels instead of coordinates to refer to cells. These labels are placed below the cells and in gray color. The faulty cell, marked with a red dashed border, is `MarriedWithHold`, which incorrectly uses `GrossPay` instead of `AdjustedWage` as the computational base. Note that the faulty cell is only used by



(a) The value view of the `Payroll` spreadsheet.



(b) The formula view of the `Payroll` spreadsheet.

Example 5.5: The modified `Payroll` spreadsheet, computing the taxes and tax exemptions for a single employee. Input cells are in blue and the result cell in purple font. The fault is in cell `MarriedWithHold`, which references `GrossPay`, but should be referencing the `AdjustedWage`.

the result cell `FedWithHold` if the person is married, i.e. `MStatus` is "Married". If we set the `MStatus` to "Single", the spreadsheet would produce the correct result, leading to coincidental correctness.

If we place a single negative testing decision in the result cell $\texttt{FedWithHold}_{\boldsymbol{\times}}$, SFL creates the CONE for this cell, comprising all six formula cells. With closer inspection of Figure 5.5b, we see that for the computation of `FedWithHold`, we only need three cells: `MStatus` to test that married status, `MarriedWithHold`, which is used when the status is not "Single", and `GrossPay`, which is referenced by `MarriedWithHold`. For this test case, it is therefore possible to include only the relevant cells in the DYNCONE, excluding `SingleWithHold` and all its subsequently referenced cells. Table 5.4 shows the computed likelihoods and the absolute and relative rank for the faulty cell. While the results are identical in the best case scenario, Dynamic Slicing reduces the number of cells left to inspect from six to two in the worst case scenario.

Table 5.4: Similarity coefficients and resulting ranks for Example 5.5 with and without Dynamic Slicing. As we use only the single failing test case $\texttt{FedWithHold}_{\boldsymbol{\times}}$, the number of failing test cases each cell participates in is equal to the SFL result using the Ochiai coefficient. The faulty cell `MarriedWithHold` is shaded in gray and its average relative rank could be improved by 33 % with Dynamic Slicing.

Cell c	$\text{SFL}_\text{O}(c)$	$\text{SFL}^d_\text{O}(c)$
`FedWithHoldAllow`	1	0
`AdjustedGrossPay`	1	0
`AdjustedWage`	1	0
`SingleWithHold`	1	0
`MarriedWithHold` (F)	**1**	**1**
`FedWithHold`	1	1
$\text{ABSRANK}_{avg}(c_\text{F})$	3.5	1.5
$\text{RELRANK}_{avg}(c_\text{F})$	58 %	25 %
$Impact_{avg}$		33 %

5.3 Tie-Breaking

Tie-breaking strategies add prioritization to critically tied cells by applying heuristics and metrics. Given a critical tie CT containing the faulty cell c_f and one or more non-faulty cells, the goal of tie-breaking is to find a function $\Omega(c)$ for each cell $c \in CT$ so that

$$\forall c \in CT, c \neq c_f : \Omega(c_f) > \Omega(c),$$

therefore positioning the faulty cell first within the tie. Since the function Ω is often based on heuristics and metrics, it is possible that the faulty cell receives a lower value than non-faulty cells, in the worst case even ranking last.

Note that any cells that are ranked higher than the critical tie will retain their ranking, and the tie-breaking strategy simply influences the order of the cells within a tie, where otherwise no prioritization would exist. In other words, a high value of $\Omega(c)$ results in a high ranking of c *within* the tie, and a low value results in a low ranking of c within the tie. This means that tie-breaking strategies cannot improve the best case scenario for the absolute rank.

We discuss both position- and metric-based tie-breaking strategies in this section. Position-based tie-breaking, as described in Section 5.3.1 ranks cells according to their absolute position in the spreadsheet or their distance to testing decisions. Section 5.3.2 focuses on metric-based tie-breaking, analyzing the cell formulas to rank the cells by complexity.

5.3.1 Position-based Tie-Breaking

In this section, we discuss several tie-breaking strategies based on a cell's position in the spreadsheet or relative position to negative testing decisions, based on the research provided by Xu *et al.* [32]. We briefly explain each of the three strategies and conclude this section by comparing the strategies and their performance when applied to Example 1.1 and the `Payroll` example from Section 5.2.2.

Cell Order Strategy (COS)

This strategy breaks ties by using the position of the cell within the spreadsheet and its distance from the first cell. We assume that cells that are far down or far right from `A1` have a higher fault likelihood, as the complexity of the spreadsheet increases with size and maintenance becomes more difficult. The original strategy, called SOS, is used for traditional programming languages [32] and uses line numbers to identify the position of the statement. However, spreadsheets are at least two-dimensional and therefore the cells do not have unique line numbers. While it is possible to assign a unique number to each cell, it would be necessary to give more weight to either column or row distance. Instead, we opt to accept a limited number of identical values for Ω and use the euclidean distance to compute cell positions, which is

$$\eta(p, q) = \sqrt{(\varphi_x(p) - \varphi_x(q))^2 + (\varphi_y(p) - \varphi_y(q))^2}.$$

Table 5.5 shows the first cells in the spreadsheet and their respective positions, using the function

$$\Omega_{\text{COS}}(c) = \eta(\texttt{A1}, c),$$

with `A1` being the first cell in the spreadsheet.

Table 5.5: Euclidean distance from cell `A1` for the cells in a spreadsheet.

	A	B	C	D	...
1	0	1	2	3	
2	1	1.4	2.24	3.16	
3	2	2.24	2.83	3.61	
4	3	3.16	3.61	4.24	
...					

We see that even with such a small number of cells, all values not in the diagonal occur twice and with increasing size of the spreadsheet, more cells may share the same distance to the top-left cell `A1`. Additionally, a workbook may contain more than one worksheet, further increasing the possible size of the critical tie. It is unclear how multiple source files are handled in the original strategy. While it would be possible to penalize multiple worksheets, we assume the distance is computed from the first cell in each worksheet, disregarding the worksheet position. As opposed to the original strategy, COS may not be able to break all ties, but it is an adequate approximation, based on the assumption that with increasing complexity of the spreadsheet, the likelihood for faults also increases.

Cell Distance Strategy (CDS)

One possible improvement over COS sets cells in relation to the reported testing decisions instead of the first cell of the spreadsheet. The concept is closely related to manual debugging, where a person may start by inspecting the cell with the negative testing decision and then fan out, inspecting closer cells first. We therefore define

$$\Omega_{\text{CDS}}(c) = -\min(\{\eta(o, c) \mid o \in \textsc{Cells} : d(o) \in TD^{-}; c \in \textsc{Cone}(o)\})$$

where we select the closest cell with a negative testing decision that c contributes to and return its negative distance to the cell c. For the distance function $\eta(o, c)$, we again use the euclidean distance, although any other coordinate-based distance formula would achieve similar results.

Path Length Strategy (PLS)

Rather than using the euclidean distance, we can also measure distance as the number of references that need to be followed from o to reach c. We define the strategy function as

$$\Omega_{\text{PLS}}(c) = -\min(\{\text{minPathLength}(o, c) \mid o \in \textsc{Cells} : d(o) \in TD^{-}; c \in \textsc{Cone}(o)\}),$$

with minPathLength being a function that provides the shortest path in the Program Dependency Graph (PDG) from cell o to cell c. Following the arrows in the PDG, we get $\text{minPathLength}(o, c) = 1$ if the cell c is referenced by the output cell o directly, $\text{minPathLength}(c) = 2$ if the cell is referenced by a cell with $\text{minPathLength}(c) = 1$ and so on. The aim of this strategy is to again follow the natural debugging process, starting from the output cell, i.e. $c = o$. Cells that are far away from the testing decision receive a lower rank than others. With this strategy, it is likely that large ties remain, as many cells may have the same path length to their output cells.

Examples

We apply the position-based strategies presented above to Example 1.1 and the `Payroll` example introduced in Section 5.2.2. For both examples, we compute the Ω function and the resulting rank for all cells in the critical tie produced by SFL. We then apply the metrics defined in Section 2.2.4 to measure the effect of the proposed techniques.

Table 5.6 shows the results for Example 1.1, which has an initial critical tie $CT = \{$`D2`, `E2`, `D4`, `E4`, `E5`$\}$ containing five elements. The table displays the Ω values for all cells in CT as well as their resulting rank within the critical tie. The results for the faulty cell `D2` are highlighted with gray shading. The table also shows the size of the resulting critical tie and the *Tie-Reduction* metric, indicating by how much the size of the critical tie could be reduced. The *Impact* metric shows the positive or negative effect these strategies have on the average relative rank of the faulty cell `D2`.

Table 5.6: Position-based tie-breaking results for Example 1.1, including the *Tie-Reduction*, ranking results and *Impact*. The results for the faulty cell `D2` are shaded in gray.

Cell c		Ω_{COS}	Rank_{COS}	Ω_{CDS}	Rank_{CDS}	Ω_{PLS}	Rank_{PLS}
`D2`		**3.2**	**5**	**-3.2**	**5**	**-2**	**3**
`E2`		4.1	4	-3.0	4	-1	2
`D4`		4.2	3	-1.4	3	-2	3
`E4`		5.0	2	-1.0	2	-1	2
`E5`		5.7	1	-0.0	1	-0	1
Critical Tie Size	5		1		1		2
Tie-Reduction			80 %		80 %		60 %
$\text{AbsRank}_{avg}(c_F)$	3.0		5.0		5.0		4.5
$\text{RelRank}_{avg}(c_F)$	25 %		42 %		42 %		38 %
$\textit{Impact}_{avg}$			-17 %		-17 %		-13 %

The high *Tie-Reduction* and low Critical Tie Size indicates that the strategies are successful at breaking the tie almost completely. However, the impact of these strategies is negative, meaning that they do not improve the average case scenario produced by SFL alone. The faulty cell `D2` is located in the last tie, meaning that the worst case scenario could not be improved. In this case, larger ties result in lessened negative impact, which is the case for PLS, as we compare the average ranks rather than the worst case ranks.

Table 5.7 shows the same strategies applied to the critical tie of the `Payroll` example, which has an initial size of six cells. In contrast to Example 1.1, the strategies show a positive impact on the `Payroll` example, meaning the $\text{RelRank}_{avg}(c_F)$ could be improved by 16 to 20 %. These results show that position-based tie-breaking strategies rely on specific structures in spreadsheets, which cannot be guaranteed and may lead to a worse ranking if not present.

Table 5.7: Position-based tie-breaking results for the `Payroll` example, including the *Tie-Reduction*, ranking results and *Impact*. The results for the faulty cell `MarriedWithHold` are shaded in gray.

Cell c		Ω_{COS}	Rank_{COS}	Ω_{CDS}	Rank_{CDS}	Ω_{PLS}	Rank_{PLS}
`FedWithHoldAllow`		4.1	6	-5.0	6	-3	4
`AdjustedGrossPay`		5.0	5	-3.6	4	-3	4
`AdjustedWage`		6.4	4	-3.0	3	-2	3
`SingleWithHold`		7.1	3	-4.0	5	-1	2
`MarriedWithHold (F)`		**7.6**	**2**	**-2.0**	**2**	**-1**	**2**
`FedWithHold`		8.6	1	-0.0	1	-0	1
Critical Tie Size	6		1		1		2
Tie-Reduction			83 %		83 %		67 %
$\text{AbsRank}_{avg}(c_F)$	3.5		2.0		2.0		2.5
$\text{RelRank}_{avg}(c_F)$	58 %		33 %		33 %		42 %
$Impact_{avg}$			25 %		25 %		16 %

5.3.2 Metric-based Tie-Breaking

As opposed to position-based tie-breaking, which uses the position of the cell within the spreadsheet to judge its fault likelihood, metric-based tie-breaking uses information from the cell's formula, disregarding its position or distance from testing decisions. Many metrics or code smells have been proposed [7, 12] and their implementation is often straight forward. The complexity of the formula can be measured in the form of the number of operations, operands, references or a multitude of other metrics. We select only a small number of these metrics as sample implementations to see how such metrics perform in comparison to other, more complex strategies.

Number of Operators (OP)

Based on the metric *Multiple Operations* [7, 12], this strategy counts the number of function calls (`SUM`, `AVERAGE`, etc.) as well as arithmetic operations (`+`,`-`,`*`,`/`, etc.). The resulting strategy function is

$$\Omega_{\text{OP}}(c) = \text{numOp}(c),$$

so that cells with many operations are ranked higher than others.

Number of References (REF)

Similarly to the number of operators, we measure the number of references in a formula cell c to indicate its complexity. This strategy is based on the *Multiple References* metric [12] and uses the strategy function

$$\Omega_{\mathrm{REF}}(c) = \mathrm{numRef}(c),$$

so that the use of many references increases the fault likelihood of c. Note that area references such as $c_1 : c_2$ count as one reference, regardless of the number of cells contained in the area. The function $\mathrm{numRef}(c)$ is equivalent to $|\mathrm{P}(\ell(c))|$, previously used in Section 5.1 for the grouping implementation.

Dispersion of References (DR)

Bregar [7] proposes the metric of the *Dispersion of References*, increasing the fault likelihood of a cell c that references a cell r that does not share the row nor the column coordinates with c. If they are either in the same row or the same column, the fault likelihood is not raised. For area references, we ensure that c is not contained in any dimension of the referenced area $r_1 : r_2$. Let p be a reference pointing to an area of cells $r_1 : r_2$, where $r_1 = r_2$ if p points to a single cell. We define the functions D_X, D_Y so that

$$D_X(p) = \begin{cases} 0 & \text{if } \varphi_x(r_1) \leq \varphi_x(c) \leq \varphi_x(r_2) \\ 1 & \text{otherwise} \end{cases}$$

$$D_Y(p) = \begin{cases} 0 & \text{if } \varphi_y(r_1) \leq \varphi_y(c) \leq \varphi_y(r_2) \\ 1 & \text{otherwise.} \end{cases}$$

Note that in spreadsheets, absolute cell references are often used in place of constant values, in which case a dispersed reference should not increase the fault likelihood. Consequently, we state that a dispersion of reference can only occur if the reference p is not absolute. We define our strategy function as

$$\Omega_{\mathrm{DR}} = \sum_{p \in \mathrm{P}_{\mathrm{rel}}(c)} D_X(p) \cdot D_Y(p),$$

which means that we count the number of dispersed references in a formula and increase the fault likelihood accordingly. Note that we use $\mathrm{P}_{\mathrm{rel}}$ to get all partially or entirely relative references, excluding absolute references.

Examples

Let us consider how the strategies Number of Operators (OP), Number of References (REF) and Dispersion of References (DR) perform for our examples. Table 5.8 shows the results for Example 1.1 and Table 5.9 the results for the `Payroll` example.

Table 5.8: Resulting ties and ranks for Example 1.1 for OP, REF and DR. The faulty cell `D2`, shaded in gray, is ranked highest for all three strategies.

Cell c		Ω_{OP}	Rank_{OP}	Ω_{REF}	Rank_{REF}	Ω_{DR}	Rank_{DR}
`D2`		**3**	**1**	**2**	**1**	**1**	**1**
`E2`		1	2	1	2	0	2
`D4`		3	1	2	1	0	2
`E4`		1	2	1	2	0	2
`E5`		1	2	1	2	0	2
Critical Tie Size	5		2		2		1
Tie-Reduction			60 %		60 %		80 %
$\text{AbsRank}_{avg}(c_F)$	3.0		1.5		1.5		1.0
$\text{RelRank}_{avg}(c_F)$	25 %		13 %		13 %		8 %
$Impact_{avg}$			12 %		12 %		17 %

Table 5.9: Metric-based tie-breaking for the `Payroll` example. The faulty cell `MarriedWithHold`, shaded in gray, ranks highest for OP and DR. Note that DR has not broken any ties and REF places the faulty cell in the last of three ties.

Cell c		Ω_{OP}	Rank_{OP}	Ω_{REF}	Rank_{REF}	Ω_{DR}	Rank_{DR}
`FedWithHoldAllow`		1	4	1	3	0	1
`AdjustedGrossPay`		3	2	3	1	0	1
`AdjustedWage`		1	4	2	2	0	1
`SingleWithHold`		4	1	1	3	0	1
`MarriedWithHold (F)`		**4**	**1**	**1**	**3**	**0**	**1**
`FedWithHold`		2	3	3	1	0	1
Critical Tie Size	6		2		3		6
Tie-Reduction			67 %		50 %		0 %
$\text{AbsRank}_{avg}(c_F)$	3.5		1.5		5.0		3.5
$\text{RelRank}_{avg}(c_F)$	58 %		25 %		83 %		58 %
$Impact_{avg}$			33 %		-25 %		0 %

For these examples, OP succeeds at breaking the critical tie with a *Tie-Reduction* between 60-67 %. OP can also improve the average rank, showing a positive *Impact*. While REF reduces the size of the critical tie to at least half the previous size, this strategy has a negative *Impact* on the `Payroll` example. This is due to the low Ω value of the faulty cell, indicating that a large number of references does not guarantee high fault likelihood. DR has the most conflicting results, with a high *Tie-Reduction* and positive *Impact* on Example 1.1 and no *Tie-Reduction* for the `Payroll` example. These results show that this strategy is highly specialized, trying to detect the specific fault of a shifted cell reference, which is present in Example 1.1 but not in `Payroll`. However, this lack of *Tie-Reduction* is not necessarily a failure of the strategy. Note that while DR has no positive *Impact* on the rank of the faulty cell, it also has no *negative Impact*, as opposed to REF. This strategy may be applied as a precursor, before other, more drastic, tie-breaking strategies are used. It offers low risk and a potentially high reward if the spreadsheet contains this specific fault.

Long Calculation Chain

Based on metrics that indicate how many cells or computation steps are needed to calculate the value of a formula [12, 7], we implemented two metrics:

1. **Cone Size (CS)**: This metric is based on the number of cells a given cell c is dependent on, meaning the cells that are needed to compute the value of c. The resulting strategy assumes that a cell with a large Cone size is more likely to be influenced by a faulty cell, whereas a cell with a small number of dependencies has a higher likelihood to contain the fault itself. The resulting Ω function is defined as
$$\Omega_{\text{CS}}(c) = -|\text{Cone}(c)|.$$
2. **Cone Level (CL)**: The size of Cone(c) may be large, especially when cell areas are referenced. The second interesting metric is therefore the maximal path length in the PDG that needs to be followed to compute the value of c, indicating the number of references that need to be followed to be able to debug the value in c. Again, we assume that a long path length indicates a lower likelihood for the cell itself to be faulty, resulting in the strategy function
$$\Omega_{\text{CL}}(c) = -\operatorname{maxPathLength}(c).$$

Note that these two metrics could just as easily be inverted: It can also be argued that a cell that is influenced by many cells is more likely to contain the fault, as the maintainability of the formula is hindered. How these two strategies perform is shown in Table 5.10 for Example 1.1 and Table 5.11 for the `Payroll` example. The tables show very little distinction between the two approaches, with CS outperforming CL by a small margin. Both approaches show a *Tie-Reduction* of around 50 % and rank the faulty cell at the highest position in both examples.

Table 5.10: Tie-breaking using CS and CL for Example 1.1, with the faulty cell `D2` shaded in gray. The faulty cell is ranked at the highest position for both strategies. CS has a slightly higher *Tie-Reduction* and *Impact*, reducing the size of the critical tie from five to two.

Cell c		Ω_{CS}	Rank_{CS}	Ω_{CL}	Rank_{CL}
`D2`		**-2**	**1**	**-1**	**1**
`E2`		-4	3	-2	2
`D4`		-2	1	-1	1
`E4`		-3	2	-1	1
`E5`		-10	4	-3	3
Critical Tie Size	5		2		3
Tie-Reduction			60 %		40 %
$\text{AbsRank}_{avg}(c_F)$	3.0		1.5		2.0
$\text{RelRank}_{avg}(c_F)$	25 %		13 %		17 %
$Impact_{avg}$			12 %		8 %

Table 5.11: Tie-breaking using CS and CL for the `Payroll` example with the faulty cell `MarriedWithHold` shaded in gray. The faulty cell ranks highest for both strategies, which perform equally well.

Cell c		Ω_{CS}	Rank_{CS}	Ω_{CL}	Rank_{CL}
`FedWithHoldAllow`		-1	1	0	1
`AdjustedGrossPay`		-1	1	0	1
`AdjustedWage`		-3	2	-1	2
`SingleWithHold`		-4	3	-2	3
`MarriedWithHold (F)`		**-1**	**1**	**0**	**1**
`FedWithHold`		-6	4	-3	4
Critical Tie Size	6		3		3
Tie-Reduction			50 %		50 %
$\text{AbsRank}_{avg}(c_F)$	3.5		2.0		2.0
$\text{RelRank}_{avg}(c_F)$	58 %		33 %		33 %
$Impact_{avg}$			25 %		25 %

6 Evaluation Corpora

To evaluate our improvements we need test data with known faults, allowing us to empirically measure and summarize the performance of our strategies. In this work, we use spreadsheet corpora which consist of spreadsheets and corresponding documentation regarding their faults and testing decisions. Ideally, a spreadsheet testing corpus consists of many spreadsheets that are diverse in complexity, size and structure. Additionally, the faults in these spreadsheets are authentic and documented and there exist authentic and documented testing decisions made by end-users, allowing for an automated batch evaluation. While these factors present the ideal testing scenario for fault localization algorithms, such an extensive spreadsheet corpus does not currently exist. In this evaluation, we use several corpora, each with its own features and shortcomings. This enables us to highlight certain aspects of our algorithms and allows a more detailed comparison between the proposed strategies. Note that our proposed improvements often have structural requirements, such as identical and adjacent formulas for grouping or conditional structures with references for dynamic slicing. When such structures are absent, these techniques will either show no effect at all or worsen the previous result. It is therefore necessary to analyze the corpora beforehand to explain which approaches are especially suited to evaluate a corpus.

In this section, we describe the used corpora in more detail, where the data originates from and how they were created. We discuss the Euses corpus in Section 6.1, Info1 in Section 6.2 and Burnett in Section 6.3. Section 6.4 gives a detailed comparison and overview of the volume and structures exhibited by each corpus.

Structure of Test Data

To automate the evaluation of the corpora, we must also automate the process to provide the testing decisions and the documentation of the faulty cells. To this end, a debugging session is simulated by using a test set, consisting of the faulty spreadsheet and a property file containing testing decisions as well as any faulty cells. Figure 6.1 shows an example for such a property file, describing the fault and testing decisions for Example 1.1. The property file provides a relative path to the spreadsheet, two positive and one negative testing decision as well as one documented fault and its fault type.

The following properties may occur in a property file:

- `EXCEL_SHEET` specifies the path to the spreadsheet for the test case, used only once per property file.
- `INCORRECT_OUTPUT_`i specifies a negative testing decision, where i is a sequence number starting from 1, and the cell coordinates are provided in the format

```
EXCEL_SHEET=..\\xls\\bonus_1FAULTS_FAULTVERSION1.xls

CORRECT_OUTPUT_1=0!E!3
CORRECT_OUTCELL_ORACLE_MISTAKE_1=F
CORRECT_OUTPUT_2=0!C!5
CORRECT_OUTCELL_ORACLE_MISTAKE_2=F

INCORRECT_OUTPUT_1=0!E!5
INCORRECT_OUTCELL_ORACLE_MISTAKE_1=F
INCORRECT_OUTCELL_EXPECTED_VALUE_1=874.0

FAULTY_CELLS_1=0!D!2
FAULT_TYPE_1=CRS
```

Figure 6.1: Property file for Example 1.1, with two positive and one negative testing decision as well as one fault. None of the testing decisions are oracle mistakes.

`<worksheet number>!<column>!<row>`.

- `INCORRECT_OUTCELL_ORACLE_MISTAKE_`i defines whether the testing decision with the sequence number i is an oracle mistake. If the value of this property equals "`T`", the testing decision should be positive instead of negative.
- `INCORRECT_OUTCELL_EXPECTED_VALUE_`i specifies the expected value for the incorrect cell with the sequence number i. This property is necessary for model-based approaches, but not required for trace-based approaches such as SFL.
- `CORRECT_OUTPUT_`i specifies a positive testing decision, with i being a sequence number starting from 1. Cells are specified in the same format used for incorrect output cells. Note that in contrast to incorrect cells, no output value is expected, as the cell produces the expected value.
- `CORRECT_OUTCELL_ORACLE_MISTAKE_`i defines whether the testing decision with the sequence number i is an oracle mistake. If the value of this property equals "`T`", the testing decision should be negative instead of positive.
- `FAULTY_CELLS_`i specifies that the formula in the cell is not correct, where i is a sequence number starting with 1 and the cell coordinates are provided in the same format as for testing decisions.
- `FAULT_TYPE_`i specifies the fault type for the faulty cell with the sequence number i, based on the mutation operators for spreadsheets specified by @Abraham2009. This property value is optional.

Each spreadsheet corpus consists of a collection of such test sets. To enable faster distinction between test cases, both the property files and the faulty spreadsheet follow a naming convention. The name consists of the spreadsheet's original name, the number

of faults (`<X>FAULTS`) and the fault version (`FAULTVERSION<Y>`), separated by an underscore. The same base spreadsheet may have several faulty versions and therefore provide multiple test sets.

6.1 EUSES

The first corpus used in our evaluation is the EUSES spreadsheet corpus [10], which is a public resource containing a wide variety of spreadsheets. This corpus has been used previously in our evaluations [15] and it contains many spreadsheets with varying sizes and structures. The corpus, as presented by Fisher and Rothermel [10], does not contain any documented faults or testing decisions. Consequently, to use this corpus, we injected the spreadsheets with artificial faults via mutation. For each of the around 1 400 base spreadsheets, we created up to five first-order mutants by using mutation operators for spreadsheets [3]. This means that each spreadsheet was injected with a single fault by changing one element in one formula of the spreadsheet at random. Due to missing input values and coincidental correctness, some of these generated faults did not cause an observable erroneous behavior in the spreadsheet, in which case they were excluded from the corpus. Finally, about 570 single-fault mutants with up to 10 000 formula cells remained for the evaluation of SFL and its improvements.

Testing Decisions

The testing decisions for this corpus were created by comparing the mutated version of a spreadsheet with the original spreadsheet. All result cells that differed from the original were marked as erroneous, while all result cells that were identical to the original were marked as correct. The number of erroneous result cells is surprisingly low, with a median of one and a maximum of 72. The number of correct result cells is significantly larger, with a median of 28 and a maximum of almost 3 000 cells, which would not be provided by a user realistically. As these testing decisions are well-documented, it is also possible to use only a subset of positive testing decisions to simulate a more realistic testing environment. Please note that the CONEs of these testing decisions might not intersect with the CONEs of the negative testing decisions at all, therefore not contributing to the test case. There currently does not exist any documentation of these kinds of testing decisions.

6.2 Info1

The second spreadsheet corpus in our evaluation, INFO1, is based on student submissions. We received access to a collection of spreadsheets, based on to four distinct assignments that were completed by civil engineering students in an Excel course. The aim of the course was to familiarize students with the way spreadsheets can be used for simple structural engineering calculations.

For each assignment, we were provided with a sample solution and around 100 student submissions. We selected two of the four tasks and created test sets with two different input sets per assignment. One exercise is the statics computation for single-span beams, the other the calculation of the bending moment for a two-span beam.

Both of these spreadsheets are adjustable, meaning that they contain a small number of input variables that can be set by the user. The computations used in these spreadsheets contain complex operations, such as long formulas with many references and/or operations. Both spreadsheets also have a high amount of copied formula cells that can be grouped. The total number of formula cells is between 500 and 3 000 per spreadsheet.

To find the authentic faults the students made while completing the assignment, the student submissions need to be compared to the sample solution. Each student created the spreadsheet from the ground up with no structural requirements and without using any templates, making direct spreadsheet comparison difficult. However, as part of the assignment, students were given a printed version of the solution, providing expected values for one specific test case. This means that while the exact position of the cells varies, most submissions exhibit similar patterns and labels.

To create test sets from these submissions, we need to be able to compare student submissions to the sample solution. To this end, we label as much of the spreadsheet as possible, looking above or left to a cell or an identical group of cells to find possible names for the cells. The found labels are compared to the sample solution, using a mapping process that allows greater variance in labels, as strings may contain spelling errors or other slightly differing formulations. For example, one student might choose to label a cell containing a length parameter as "L", another as "L=" and yet another as "length:". Such labels must be analyzed manually and mapped to the same identifier, for example "Length". Note that the grouping implementation discussed in Section 5.1 was used to identify groups of cells with identical formulas, which usually share the same label.

The labeling of the cells allows a limited comparison to the sample solution, comprising only the cells whose label occurs in both the sample solution and the student submission. For the creation of the test set, all input cells from the sample solution need to be labeled in the submission as well as at least one result cell to compare the computed values. To create variants of the same spreadsheet with different values, input values are changed in the sample solution and copied to the submission, provided all input labels could be found in the submission. We reevaluate the submission spreadsheet and compare its values to the ones in the sample solution for each matched label. If the values produced by the submission differ from the one in the sample solution, we have an erroneous behavior and the spreadsheet is considered faulty. We then manually analyze the faulty spreadsheet with the aid of existing fault localization techniques in order to locate authentic faults.

As is the case in our Example 2.1, a single fault may affect an entire group of cells, as the fault in the first cell is copied down to the adjacent cells. For this corpus, many such faults occurred, resulting in a high number of faults that can be found by the fault localization technique. As all of these faults can be fixed if one of them is found, we treat these faults as single faults for the comparison with Euses and Burnett, but mark each individual cell as faulty for the evaluation, allowing the Sfl algorithm to find the best ranked cell.

Testing Decisions

Once the faults are documented for each test set, we create artificial testing decisions by reusing the compared values. As we have access to a sample solution, many correct testing decisions are found. Due to the use of grouping, we often have testing decisions for an entire group of cells, which may contain more than 100 cells in these spreadsheets. Naturally, this volume of testing decisions is unlikely to be supplied by the end-user. We therefore select only a subset of these testing decisions, simulating a more realistic debugging environment. Similarly to the testing decisions created for EUSES, we supply testing decisions for all result cells, ensuring the broadest possible test coverage. However, to reduce the amount of testing decisions, we select only one testing decision per group, always selecting the last cell to ensure consistent test cases.

We allow testing decisions to be set in any formula cell in the spreadsheet and testing decisions in non-result cells are a requirement for the Blocking technique to have any effect. We therefore also allow a small set of optional testing decisions, selected from the remaining non-result formula cells. Again, we allow only one testing decision per group by always choosing the last cell. The number of optional testing decisions is chosen randomly with upper limits to ensure a realistic amount of testing decisions. The upper cap for all testing decisions is 10 for negative and 20 for positive testing decisions. These limits can only be exceeded by testing decisions for result cells, in which case no optional testing decisions are added.

6.3 Burnett

To compare our results to previous evaluations, we requested the experiment data used by Ruthruff *et al.* [27]. In their user study, they asked 20 participants to debug spreadsheets using their testing methodology WYSIWYT as well as the feedback from their fault localization technique, Nearest Consumer, which is described briefly in Section **??**.

6.3.1 User Study

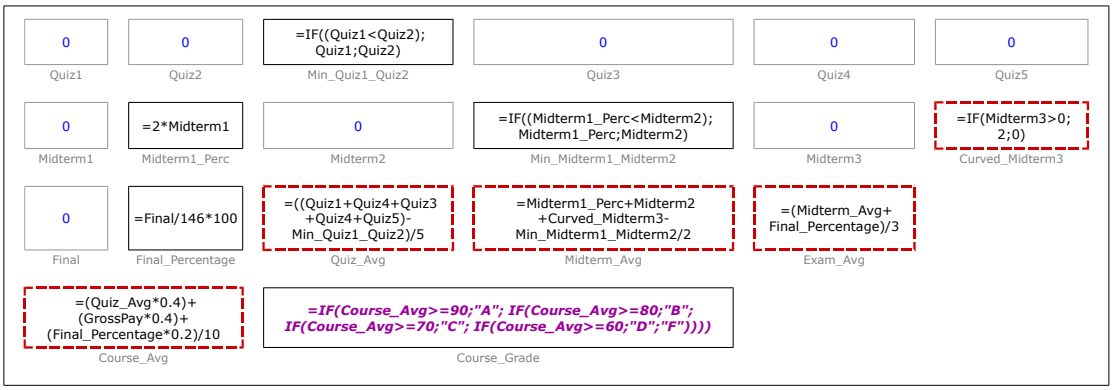
We first describe the setup of the experiment in more detail, including the two spreadsheets and the general process of the study.

Setup

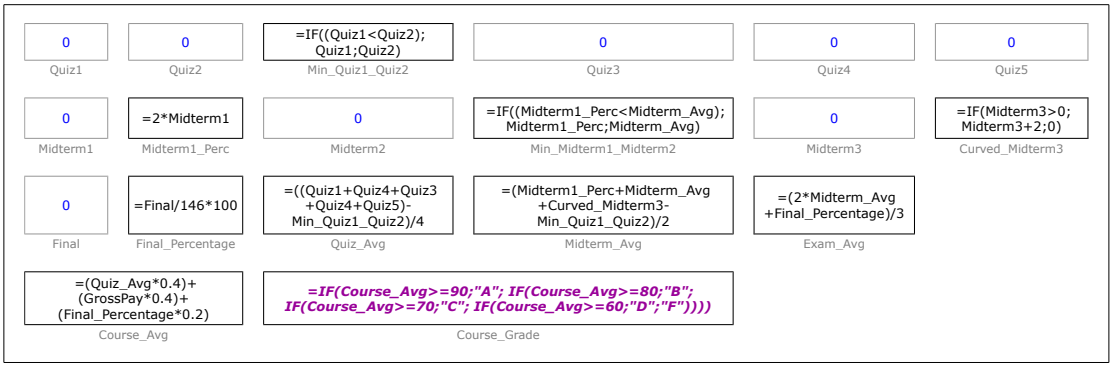
After executing a separate user study to find authentic and user-produced faults, the researchers created two separate spreadsheets, `Gradebook` and `Payroll`, which were injected with five faults each. While the injected faults are comparable to authentic faults in that their type and form has occurred before, the position and frequency in which the faults occur in the seeded spreadsheets create a simulated testing environment.

The `Gradebook` spreadsheet shows the computation of a students grade at the end of the term, including results from quizzes, midterms and an exam to calculate a course grade. This spreadsheet contains nine input cells and ten formula cells, of which four cells

contain `IF` constructs. The spreadsheet the participants were presented with contained no input values and was seeded with five faults in five different cells. Figure 6.2 shows both the faulty and the correct formula view, omitting the value view as all values were initialized to zero.



(a) The formula view of the faulty `Gradebook` spreadsheet.



(b) The formula view of the correct `Gradebook` spreadsheet.

Figure 6.2: The `Gradebook` spreadsheet from the BURNETT corpus, containing five faults.

The other spreadsheet, `Payroll`, shows a detailed computation of an employee's salary, insurance and tax costs. The slightly larger spreadsheet contains six input cells and 18 formula cells, nine of which contain `IF` statements. The faulty spreadsheet presented to participants is initialized to input values of zero and contains five faults in four cells. One cell contains two faults, one of which is classified as an omission fault. For our own evaluation, we focus only on finding the faulty cell itself and assume that once it is found, both faults can be fixed by the user. Figure 5.5 shows a modified version of this spreadsheet, showing a subset of the cells contained in the original task.

Study Process

The goal of the study was to show the effectiveness of fault localization techniques as aids to end-user debugging. Additionally to the spreadsheet, the participants received instructions and specifications with example input values and expected output values. The participants debugged the given spreadsheets by inserting input values and using the testing methodology WYSIWYT to place X and check marks in formula cells. Additionally, the participants were split into treatment and control groups, where the treatment group received fault localization feedback as soon as negative testing decisions were set, whereas the control group relied simply on the testing feedback. All of the user's actions, meaning input and formula changes as well as testing decisions, were written to log files, allowing for further analysis of the user data.

6.3.2 Conversion to Local Format

The spreadsheets Ruthruff *et al.* [27] used in their study are written in Forms/3, a spreadsheet language which separates cells from the tabular matrix form, allowing end-users to place cells without grid restrictions. The focus of this spreadsheet language is the readability of the cells and cell references, labeling each cell and using cell names rather than coordinates to refer to cells in formulas. To allow portability, we converted the spreadsheets from Forms/3 to Microsoft Excel 1997 files with the extension `.xls` using Apache POI[1]. The cell names were retained by using named ranges.

As described in Section 6.3.1, the provided testing decisions and user inputs are documented in log files. To create test sets in our specified format, we convert the logs to `.properties` files. We distinguish between four user actions:

1. **Check mark:** The user places a positive testing decision in a formula cell, judging its value to be correct.
2. **X mark:** The user places a negative testing decision in a formula cell, judging its value to be incorrect.
3. **Input value changed:** The user provides a different input value, creating a new test set. Previous testing decisions regarding this cell and its consumers are removed.
4. **Formula changed:** The user changes a formula cell, presumably fixing the erroneous behavior. Previous testing decisions regarding this cell and its consumers are removed.

Note that since these inputs are created by humans, they may contain errors. Users that are not familiar with spreadsheet syntax might make errors when changing formulas, resulting in formulas that cannot be parsed. It is also possible that the user misjudges values, setting incorrect testing decisions. Ruthruff *et al.* [27] also state that some users understood the testing decisions as judgment regarding a cell's *formula* instead of its value, changing the nature of the testing decision completely.

[1] http://poi.apache.org/spreadsheet/index.html

Conversion

To make use of all available user input, we create test sets which offer different input values and suitable testing decisions as well as the real number of faults remaining in the spreadsheet. In this case, we need both the faulty spreadsheet the users are presented with and its corrected version, which the users aim to achieve. We track the changes the user makes to the spreadsheet, modifying inputs and formulas, and one or more modified spreadsheets are created for each session. Once the user changes a formula, we compare the modified spreadsheet to the correct version, finding differences in the formulas and extracting the remaining faulty cells. Note that it is also possible that the user adds a fault by mistake. These faults can also be tracked, therefore reaching a higher number of faults. Additionally, expected output values for cells with X marks can be extracted by comparing the faulty and the corrected spreadsheet.

The number of resulting test sets is dependent on the break points we use to save the data set. The smallest possible unit, where each testing decision results in a new test set, is not efficient: The user provides testing decisions sequentially and several testing decisions might be needed to track down a single fault. We therefore create a test set each time the user changes either an input cell or a formula cell in the spreadsheet, which can be seen in line 13 in Algorithm 2. The procedure SAVETESTSET creates a test set from the provided user input, and if necessary compares the working copy of the spreadsheet with the correct version. It is possible to extract both faults and testing decision in this manner, depending on the requirements. The different options for the testing decisions and the origin and complexity of the faults can be seen in Table 6.1. Additionally, it is possible to filter out test sets where the user only provided X marks for faulty cells, as the potential of fault localization is hardly used in those cases.

Formula or input changes provide the latest possible point of conversion, as the testing decisions are removed for the modified cell c and all its consumers. While WYSIWYT supports multiple input values, allowing several test cases for the same cell to influence the fault likelihoods, our system does not currently support such test sets. It is therefore necessary to create a separate test set for any new input values. If the user changed a formula cell rather than an input cell, we assume that the fault was fixed. Note that a new test set is only valid if there exists at least one negative testing decision, as fault localization would not be possible otherwise.

As we are able to compare the working copy of the spreadsheet with a corrected version, we can detect oracle mistakes in formulas and testing decisions as well as coincidental correctness. As we want to experiment with real user data, it makes sense to document these errors and evaluate them separately. Additionally, we can use the user input given by input changes to create our own testing decisions for each result cell, comparing the values of the working copy with the corrected version. This is comparable to how testing decisions are created for the EUSES corpus.

A single created test set contains a spreadsheet with the provided inputs and a property file, containing all faulty cells and their fault types, one or more negative testing decisions with expected output values and, optionally, any number of positive testing decisions.

Algorithm 2 Logfile Conversion

Input: logfile L, faulty spreadsheet S_f, correct spreadsheet S_c
Output: TS=$\{\text{t} : \text{t} = \{S_f, TD^-, TD^+, F\}\}$

```
1: init TS, TD+, TD−
2: while read line l from L do
3:     a ← getAction(l)                          ▷ get relevant information
4:     c ← getCell(l)
5:     if a == Testing Decision then
6:         v ← getValue(l)
7:         if v is negative then                 ▷ add testing decision
8:             TD− ← TD− ∪ d(c)
9:         else
10:            TD+ ← TD+ ∪ d(c)
11:        end if
12:    else if a == Formula Change then
13:        saveTestSet(TD−, TD+)                 ▷ save the test set
14:        e ← getExpression(l)
15:        setFormula(c,e)                       ▷ handle the user input
16:        for all c̃ ∈ consumers(c) do           ▷ remove testing decisions
17:            TD− ← TD− \ d(c̃)
18:            TD+ ← TD+ \ d(c̃)
19:        end for
20:    end if
21: end while
22: saveTestSet(TD−, TD+)                        ▷ save final test set

23: procedure saveTestSet(TD−, TD+)
24:    F ← getFaultyCells(Sc, Sf)                ▷ compare spreadsheets
25:    TD− ← checkTDs(Sc, Sf, TD−)               ▷ check for user errors
26:    TD+ ← checkTDs(Sc, Sf, TD+)
27:    TS ← TS ∪ {Sf, TD+, TD−, F}               ▷ save test set
28: end procedure

29: procedure setFormula(Cell c, Expression e)
30:    if e = κ then                             ▷ expression is a constant
31:        Sf ← setValue(Sf, c, e)               ▷ new input value
32:        Sc ← setValue(Sc, c, e)
33:    else
34:        Sf ← setFormula(Sf, c, e)             ▷ possible fix for faulty spreadsheet
35:    end if
36: end procedure
```

Table 6.1: Options for the conversion of logfiles.

Fault Complexity	Fault Origin	Testing Decisions
• single • single if possible, else multiple • multiple	• documented faults only • documented faults and additional faults • differences to correct version	• user provided TDs (may contain errors) • only correct TDs ($\subseteq TD$, no errors) • user provided TDs with corrected values ($= \lvert TD \rvert$, no errors) • result values

6.4 Corpora Comparison

In this section, we provide a comparison of the structure and properties of all corpora. We consider both the volume, i.e. the number of spreadsheets contained in a corpus, as well as the size of these spreadsheets. Additionally, we provide statistics regarding their suitability for dynamic slicing, grouping and tie-breaking strategies.

6.4.1 Spreadsheet Properties

For SFL to function, a spreadsheet must exhibit certain structural properties, such as the existence of cell references. A spreadsheet containing only input cells could not be debugged with SFL, as there are no data dependencies and therefore all cells are equally likely to be faulty. The improvements formulated in this work also utilize various structures that are not used by all spreadsheets. We therefore analyze the corpora regarding their structural features in the following paragraphs.

Corpus Volume

First, we compare the volume of the spreadsheet corpus, i.e. the number of spreadsheets each corpus contains. For testing purposes, there usually exist correct base spreadsheets and several faulty versions of them, containing one or more faults. Table 6.2 compares the corpora regarding their number of contained spreadsheets as well as the size of these spreadsheets. A large number of base spreadsheets indicates a wide variety and diversity between spreadsheets. The EUSES corpus has the widest range of spreadsheets, containing 184 base spreadsheets and 576 faulty spreadsheet versions or test sets. It also has the most variance between the spreadsheets, containing both the smallest and the largest spreadsheets of all three corpora. While both INFO1 and BURNETT are based on two spreadsheets, the spreadsheets in the INFO1 corpus are student submissions modeling a sample solution, whereas the spreadsheets in BURNETT were provided by the researchers and only the corrections to the spreadsheets are created by the study participants. The

spreadsheets in the BURNETT corpus are also noticeably smaller than the spreadsheet in the other corpora with an average of only 15 formula cells per spreadsheet.

Table 6.2: Comparison of corpus volume, indicating both the number of correct base spreadsheets and faulty versions. The size of the spreadsheets themselves is indicated by the number of formula cells.

Feature		EUSES	INFO1	BURNETT
Number of base spreadsheets		184	2	2
Number of faulty spreadsheet versions		576	119	349
Number of formula cells	Min	6	501	10
	Q1	40	580	12
	Median	111.5	2131	18
	Q3	305.25	2245.5	19
	Max	10316	3157	19
	Average	353.95	1466.22	15.03
Number of input cells	Min	1	10	5
	Q1	45	13	5
	Median	129	21	6
	Q3	430	60	9
	Max	24067	733	9
	Average	601.54	90.67	7.17

Grouping

For any type of grouping to have an effect, a spreadsheet needs groupable cells. To indicate how suitable a corpus is for grouping, we compare the number of formula cells in a spreadsheet to the number of unique formulas. Let F be the set of all formula cells in a spreadsheet, so that

$$F = \{\forall c \in \text{CELLS} : \rho(\ell(c)) \neq \varnothing\}.$$

We then have a subset of unique formula cells, F_U, so that

$$F_U \subseteq F : \forall c, c' \in F_U, c \neq c' : \ell(c) \neq \ell(c').$$

Finally, we compute the percentage of copied formula cells as

$$P_C = \left(1 - \frac{|F_U|}{|F|}\right) \cdot 100,$$

where a high percentage indicates that Grouping is well suited for this corpus. Table 6.3 shows the average and median values for $|F|, |F_U|$ and P_C for all corpora. Both EUSES and INFO1 show high percentages of copied formulas. Due to the small number of cells in BURNETT, there are no copied formulas in this corpus and grouping will have no effect on these spreadsheets.

Table 6.3: Comparison of code duplicity, indicating the number of copied formulas. A high percentage of copied formulas indicates that the Grouping technique is suitable for this corpus.

Feature		Euses	Info1	Burnett
Number of formula cells	Min	6	501	10
	Q1	40	580	12
	Median	111.5	2131	18
	Q3	305.25	2245.5	19
	Max	10316	3157	19
	Average	353.95	1466.22	15.03
Number of unique formulas	Min	2	13	10
	Q1	6	21	12
	Median	11	23	18
	Q3	26	28	19
	Max	895	90	19
	Average	24.19	25.91	15.02
Percentage of copied formulas	Min	0 %	84 %	0 %
	Q1	75 %	96 %	0 %
	Median	89 %	98 %	0 %
	Q3	95 %	99 %	0 %
	Max	100 %	99 %	5 %
	Average	82 %	97 %	0 %

Dynamic Slicing

Table 6.4 compares the corpora regarding their contained `IF`-statements, where a large number of `IF`s indicates that Dynamic Slicing may be a successful improvement. The table shows that Info1 contains the largest number of `IF` statements. It should be noted that in the Euses corpus, less than 20 % of spreadsheets contain one or more `IF`s. While the Burnett corpus contains only a small number of cells and therefore a small number of `IF`s, they can be found in nearly every cell in `Payroll`. The number of cells with `IF` statements is always less than or equal to the total number of `IF`s and shows that such statements are often nested.

Table 6.4: Absolute number and percentage of spreadsheets in the corpus containing at least one `IF`-statement and of these spreadsheets, the number of cells containing `IF`s and the total number of `IF`-statements.

Feature		Euses	Info1	Burnett
Number of spreadsheets containing `IF`s		109	112	349
Percentage of spreadsheets containing `IF`s		19 %	94 %	100 %
Of those, number of cells with `IF`s per spreadsheet	Min	1	142	4
	Q1	9	143	4
	Median	49	284	9
	Q3	107	1616	9
	Max	1983	1632	9
	Average	136.38	783.25	6.52
Of those, number of `IF`s per spreadsheet	Min	1	142	7
	Q1	22	143	7
	Median	54	532.5	9
	Q3	136	1616	9
	Max	7839	3234	9
	Average	371.44	1023.28	8.01

A large number of `IF` statements per spreadsheet increases the likelihood that dynamic slicing can improve the result. It should be noted that this is merely a rough indicator, as the DynCone may not change at all if the `IF` statement does not reference other cells or the same cells in both branches. It is also possible that the cells containing the `IF` statements are not needed to produce the failing test case, in which case dynamic slicing will also have no effect on the ranking.

Spreadsheet Complexity

Table 6.5 shows the average number of operators or referenced cells per formula cell, which indicate the complexity of the spreadsheet. A high number indicates many complex formulas which are difficult to debug. Such corpora are especially useful for tie-breaking and metric-based improvements, providing a realistic debugging environment.

Table 6.5: Number of operators and referenced cells per formula cell, which indicate the complexity of the formulas contained in the corpora.

Feature		Euses	Info1	Burnett
Avg. no. of operators per formula cell	Min	0.34	2.54	2
	Q1	1	3.33	2.21
	Median	1.48	3.87	2.33
	Q3	2	5.92	3.17
	Max	17	9.25	3.9
	Average	1.89	4.47	2.72
Avg. no. of referenced cells per formula cell	Min	0.97	2.97	1.58
	Q1	1.92	3.55	1.68
	Median	2.86	3.82	1.74
	Q3	5.05	4.38	1.92
	Max	232	96.65	2.6
	Average	6.17	5.96	1.87
Max. no. of operators per formula cell	Min	1	9	4
	Q1	2	9	7
	Median	3	14	7
	Q3	6	14.5	8
	Max	123	22	11
	Average	5.55	12.86	7.18
Max. no. of referenced cells per formula cell	Min	1	6	3
	Q1	4	101	3
	Median	11	143	4
	Q3	26	145.5	6
	Max	463	2829	6
	Average	28.67	381.98	4.39

6.4.2 Test Case Properties

As the testing corpora provide both the testing decisions and the documentation of faults in lieu of the user, it is necessary to compare the origin, frequency and types of testing decisions and faults. Table 6.6 shows the origin and the number of testing decisions provided for each corpus. Only the Burnett corpus has authentic testing decisions that were provided by end-users in an experiment. For the other corpora, it is necessary to simulate testing decisions, for example by setting testing decisions for all result cells, as it is done for Euses. In this case, it is necessary to have the corrected version of the spreadsheet, whose values can be compared to the faulty version. A positive testing decision (TD^+) is reached if the values of the cell in both spreadsheets are equal. Otherwise, an unexpected value is produced, leading to a negative testing decision (TD^-). This approach potentially leads to an unrealistic number of testing decisions, where Euses provides a maximum of almost 3 000 positive testing decisions. The simulation of testing decisions was modified for Info1, so that only one testing decision per group is allowed and there exists an upper limit for testing decisions. Note that in contrast to Euses, the testing decisions for Info1 may also occur in other formula cells in addition to all result cells.

Table 6.6: Comparison of testing decision origin as well as the number of positive and negative testing decisions.

Feature		Euses	Info1	Burnett
Testing decision origin		result cells	formula cells	user provided
Number of TD^+ per test set	Min	0	1	0
	Q1	9	6	2
	Median	28	8	4
	Q3	92	10	7
	Max	2962	17	17
	Average	79.81	8.08	4.76
Number of TD^- per test set	Min	1	1	1
	Q1	1	2	1
	Median	1	3	1
	Q3	2	4	2
	Max	72	10	14
	Average	2.86	3.18	1.77

In previous sections, we discussed issues in fault localization that arise from coincidental correctness and oracle mistakes. Coincidental correctness occurs in all three corpora, the detailed distribution of which can be seen in Table 6.7. While Euses has the highest absolute number of coincidentally correct testing decisions, Info1 and Burnett have higher average and median values. The table shows that for those two corpora, over 50 % of test sets contain at least one coincidentally correct testing decision. The percentage of

coincidental positive testing decisions shows the fraction of coincidentally correct testing decision in relation to all positive testing decisions in a test set. We see that on average, this percentage is very low for EUSES and nearly 30 % for INFO1 and BURNETT.

Table 6.7: Comparison of coincidental correctness in positive testing decisions.

Feature		EUSES	INFO1	BURNETT
Number of coincidental TD^+ per test set	Min	0	0	0
	Q1	0	0	0
	Median	0	2	1
	Q3	0	4	2
	Max	41	12	8
	Average	0.64	2.29	1.36
Percentage of coincidental TD^+ per test set	Min	0 %	0 %	0 %
	Q1	0 %	0 %	0 %
	Median	0 %	27 %	23 %
	Q3	0 %	50 %	50 %
	Max	100 %	100 %	100 %
	Average	2 %	29 %	29 %

Due to the origin of the testing decisions, BURNETT is the only corpus where the testing decisions were set manually by users and may therefore contain oracle mistakes. Table 6.8 shows the distribution of oracle mistakes in positive and negative testing decisions, both the absolute numbers as well as the percentage of oracle mistakes in relation to all testing decisions of the same kind, i.e. positive or negative.

The table shows that it is possible for all testing decisions in a test set to be oracle mistakes, in which case fault localization will be severely impeded. We also see that the average number of oracle mistakes in TD^- is lower than in TD^+, which means that, overall, less mistakes are made when setting negative testing decisions. However, the median shows that over 50 % of all test sets contain at least one oracle mistake in their positive testing decisions. This means that we are more likely to encounter oracle mistakes in positive testing decisions, and supports our decision to weigh negative testing decisions stronger than positive ones.

However, the average percentage of oracle mistakes is slightly higher for negative testing decisions and the third quartile indicates that at least 25 % of test sets have negative testing decisions consisting entirely of oracle mistakes. These high percentages are likely due to the smaller total number of negative testing decisions in comparison to positive ones. If all negative testing decisions are oracle mistakes, fault localization cannot provide any meaningful results.

Please note that the average percentage of oracle mistakes in TD^- is slightly higher that for TD^+, yet the median indicates that over 50 % of test sets contain oracle mistakes in TD^+. While less test sets contain oracle mistakes in TD^-, at least 25 % of test sets have negative testing decisions that consist entirely of oracle mistakes.

Table 6.8: The number of oracle mistakes in positive and negative testing decisions for the Burnett corpus.

Feature		Absolute	Percentage
Number of oracle mistakes in TD^+ per test set	Min	0	0 %
	Q1	0	0 %
	Median	0	8 %
	Q3	1	50 %
	Max	8	100 %
	Average	0.99	26 %
Number of oracle mistakes in TD^- per test set	Min	0	0 %
	Q1	0	0 %
	Median	0	0 %
	Q3	1	100 %
	Max	8	100 %
	Average	0.62	32 %

Finally, to evaluate the success of our approaches, the provided test cases must also document the faults contained in the spreadsheet. Table 6.9 shows a comparison of the number of faults as well as the fault origin. We see that Euses and the majority of the Info1 test sets contain single faults, and at least 75 % of all test sets in Burnett contain four or more faults.

Table 6.9: Comparison of fault origin, complexity and the number of faults.

Feature		Euses	Info1	Burnett
Fault origin		injected	authentic	injected
Fault complexity		single	multiple	multiple
Number of faults per test set	Min	1	1	1
	Q1	1	1	4
	Median	1	1	4
	Q3	1	1	5
	Max	1	4	8
	Average	1	1.26	4.3

7 Evaluation

In this chapter, we evaluate the techniques proposed in Chapter 5. Section 7.1 motivates the need for improvement by showing how the established version of SFL performs on our largest and most diverse corpus. We analyze the absolute and relative ranks as well as the discrepancies between best and worst case scenario. This section shows that one of the main issues SFL faces is that of a large critical tie size. Section 7.2 compares the ranks achieved for each improvement strategy and measures the effect of our strategies with the metrics discussed in Section 2.2.4. Our results indicate that Pre-Process Grouping promises the most improvement while posing no risk to worsen the results. For tie-breaking strategies, the Dispersion of References (DR) strategy is well suited for specific test sets, providing high impact for those and posing a low risk for other test sets. The Cell Distance Strategy (CDS) also offers satisfactory improvements, but comes with an elevated risk to rank the faulty cell too low.

7.1 Analysis of the Corpus

To see which areas of fault localization need improvement, we evaluate the existing implementation of SFL on the EUSES corpus. We chose this corpus for our analysis as it spans 578 test sets and is therefore the largest and most diverse of the three corpora. In comparison, INFO1 and BURNETT contain only a small number of base spreadsheets and consequently offer little variance.

First, we inspect how many non-faulty cells are ranked higher than the faulty cell for SFL. Figure 7.1 shows a histogram of the best case absolute rank ($\text{ABSRANK}_{best}(c_f)$) for each test set in EUSES. An absolute rank of one indicates the highest position, i.e. the faulty cell is ranked first. The histogram spans a bin width of one. This figure indicates that SFL is successful at ranking the faulty cell at the highest position in 536 of 578 cases, with only a few outliers. This is emphasized by the median and mean metrics, which are indicated in the figure with bright and dark magenta lines respectively. On average, the faulty cell is ranked at the second highest position.

We previously stated in Section 4.3 that one major issue of SFL is that while the faulty cell might be ranked highest, there may be a large number of non-faulty cells receiving the same rank. In this case, we speak of a critical tie with a size significantly greater than one. We use the worst case scenario to evaluate such cases, assuming that in the worst case, the user will have to inspect all non-faulty cells featuring the same rank before reaching the faulty one.

Figure 7.2 shows the distribution of the worst case absolute rank produced by SFL for the EUSES corpus. We use a higher bin width of five with the right-closed interval $(0, 5]$, as ranks start from one rather than zero. We also use a larger x-scale for this

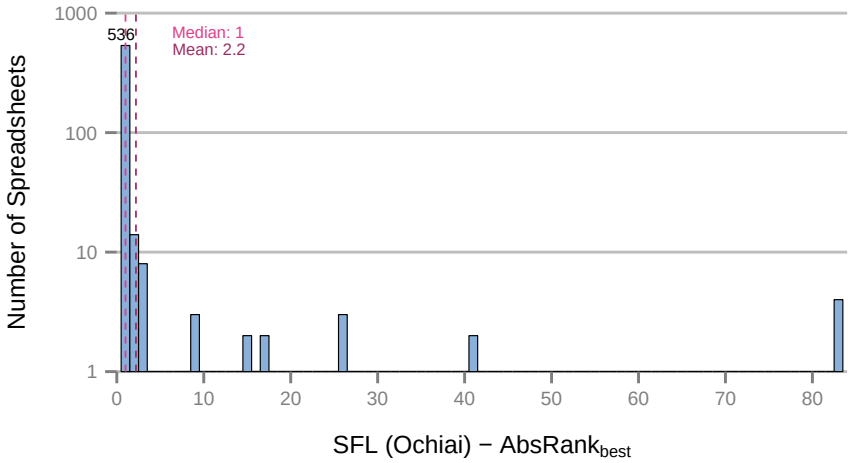


Figure 7.1: Histogram showing the best case absolute rank ($\text{AbsRank}_{best}(c_f)$) for Sfl with Ochiai on the Euses corpus, with a logarithmic scale for the y-axis. The mean and median values are in dark and bright magenta, respectively.

figure, as the resulting ranks are significantly higher. The median shows that for 50 % of all test sets, the faulty cell has a rank of five or lower. However, on average, the user needs to inspect 49 cells to reach the faulty one, a considerably larger number than for the best case scenario. There also exist three outliers with even larger ranks, which were removed from the histogram to maintain readability. These outliers reached worst case absolute ranks of over 4 500. These numbers show that large ties are indeed an issue that significantly impedes the performance of Sfl.

To set Sfl's results into context, let us consider a trivial approach to fault localization by using the union of negative Cones. The union is an unranked set of cells that represents the entire search space for each test set, including all potentially faulty cells. As the results are not ranked, all cells in the union are critically tied. This is equal to a ranking with one rank, where the faulty cell always has the highest rank and the size of the critical tie is equal to the size of the union. The worst case scenario for the union, where the user has to inspect all non-faulty cells in the critical tie before reaching the faulty cell, provides the upper limit for Sfl results and therefore gives an idea how Sfl performs in comparison. Figure 7.3 shows the histogram for the worst case absolute rank produced by the union of negative Cones, using the same scale and bin width as Figure 7.2. To ensure comparability to Figure 7.2, ten outliers reaching sizes of over 10 000 were omitted from the histogram. The dark magenta line shows that the average size of the union is twice as high as the rank produced by Sfl for the same data set. The outliers reach considerably larger values than the maximum ranks produced by Sfl. The performance of the union regarding the worst case scenario is interesting as it shows that issues with large ties are not exclusive to Sfl and also occur in other fault localization techniques.

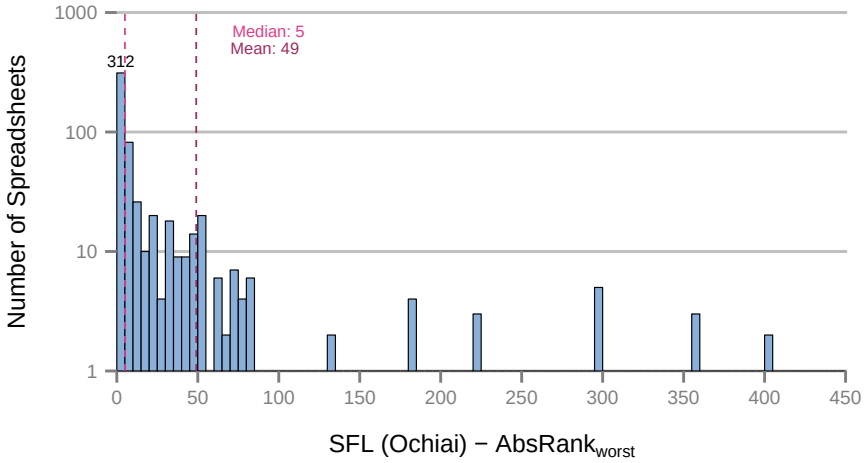


Figure 7.2: Histogram showing the worst case absolute rank ($\text{AbsRank}_{worst}(c_f)$) for SFL with Ochiai on the EUSES corpus. The mean and median values are in dark and bright magenta, respectively.

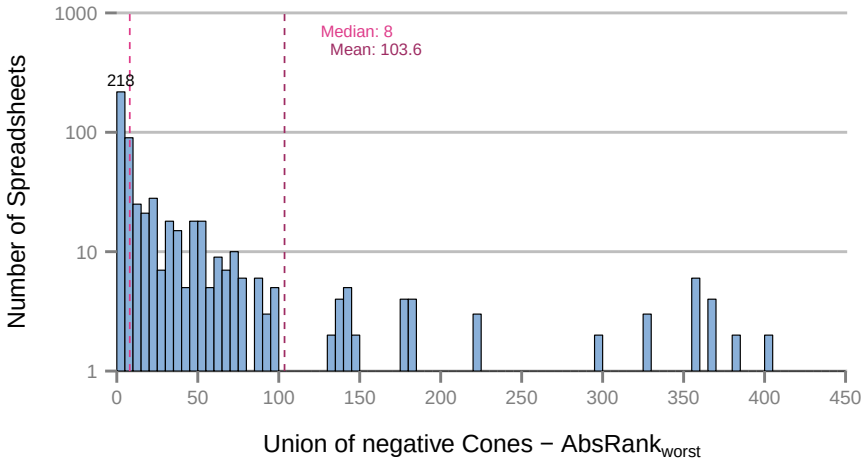
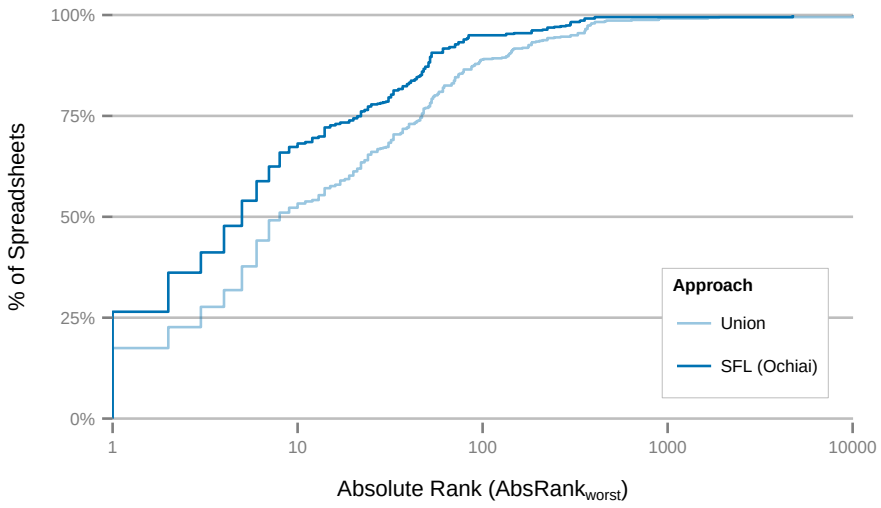


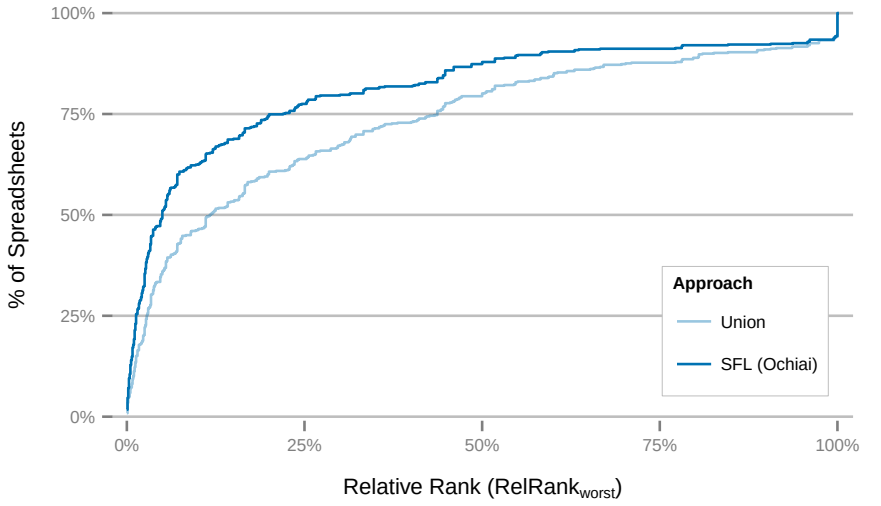
Figure 7.3: Histogram with logarithmic y-scale, showing the size of the union of negative CONEs on the EUSES corpus. The mean and median values are in dark and bright magenta, respectively.

Finally, Figure 7.4 compares the cumulative worst case ranks produced for both SFL and the union of negative CONEs in fainter color. We use the empirical cumulative distribution function, short ECDF, to allow a comparison of the two approaches and to include all outliers. The ECDF is a cumulative histogram for our empirical data. The stepping function reflects the discrete nature of the data. Figure 7.4a shows the absolute rank and Figure 7.4b shows the relative rank, where the absolute rank is set in relation to the number of formula cells. A reading example for Figure 7.4a is that over 50 % of the examined test sets have an absolute rank of five or lower. Figure 7.4b shows that, for 75 % of the test sets, we have relative rank of 25 or lower, meaning the fault is found in the first quarter of the search space.

This comparison shows that SFL consistently performs better than the union. To evaluate our approaches, it is also interesting to see whether our proposed techniques can also improve the results for the union, and if so, whether these results outperform SFL. Rather than exercising the entire evaluation for the union as well as SFL, we will highlight unexpected union performances as they occur.



(a) ECDF for the worst case absolute rank



(b) ECDF for the worst case relative rank

Figure 7.4: Empirical cumulative distribution function comparing worst case ranks produced for the union of CONEs and SFL over a logarithmic x-axis scale.

7.2 Evaluation of Improvements

In this section, we evaluate the proposed improvements on all three corpora. First, we discuss the evaluation setup in Section 7.2.1. We evaluate Grouping and the input reduction strategies in Section 7.2.2 and focus on the performance of the tie-breaking strategies in Section 7.2.3.

7.2.1 Evaluation Setup

We implemented the proposed strategies so that they can be applied to any fault localization technique that returns a ranked list of cells. For this evaluation, we apply our techniques to the SFL approach using the Ochiai coefficient and the union of negative CONEs for a rough comparison.

The BURNETT and INFO1 corpora contain multiple faults, meaning that more than one cell in the spreadsheet may be marked as faulty. To compute the average relative rank for multiple faults, we only use the rank of the highest ranked faulty cell. We assume that the debugging process is iterative, meaning that once one fault is found and fixed, the test set is run again and additional faults can be found.

Let us discuss briefly the steps needed to evaluate the performance of our improvements. First, we compute the empirical cumulative distribution function, or ECDF, for the worst case relative rank ($\text{RELRANK}_{worst}(c_f)$). This indicates how well the faulty cell is ranked in relation to all formula cells in the spreadsheet. The relative rank is meaningful, as it allows the comparison of small spreadsheets with larger spreadsheets containing many formula cells.

We also provide tables to summarize how the strategies affect the average case scenario, using the metrics defined in Section 2.2.4. We use the average case rank for these metrics, comparing the strategies to pure chance, which dictates that it is equally likely to find the faulty cell in the first half of the critical tie as it is to find it in the second half. This is vital, as most of the approaches come with the risk of possibly ranking faulty cells lower than before or placing the faulty cell last within the critical tie.

The *Impact* metric provides a risk analysis for the approach. As a starting point, we use the ranks produced for the average case scenario, where the user must inspect 50 % of the critical tie to reach the faulty cell. An *Impact* of 50 % would indicate that the approach places the faulty cell at the beginning of the tie (best case scenario). Likewise, a negative *Impact* of -50 % can be interpreted as the approach placing the faulty cell at the end of the tie (worst case scenario). Input reduction strategies and Grouping can surpass these values, effectively improving the best case rank or worsen the worst case rank. As tie-breaking works within the bounds of the critical tie, this is not possible for tie-breaking strategies.

Note that we use different scenarios for tables and figures: The figures with the ECDF assume a worst case scenario, which emphasizes even small improvements. While we apply our approaches to the union of CONEs for comparison in the figures, the tables are based on the results produced by SFL only.

Each corpus has a percentage of test sets that already have a worst case absolute rank

of one, which cannot be improved further. These test sets are pruned from our data set for the impact tables to get a more meaningful measure of the effectiveness. Table 7.1 shows the number and percentage of improvable test sets per corpus.

Table 7.1: Number of test sets that can be improved, meaning that the size of the critical tie and/or the position in the ranking is greater than one ($\text{AbsRank}_{worst}(c_f) > 1$).

Corpus	Test Sets	Improvable Test Sets	Percentage
Euses	578	425	73.5 %
Info1	102	88	86.3 %
Burnett	291	176	60.5 %

7.2.2 Grouping and Input Reduction Strategies

Let us now discuss our findings in more detail, focusing on Grouping and input reduction strategies. We evaluate the three Grouping strategies Pre-, Post- and Combined Grouping, as well as Blocking and Dynamic Slicing. Figure 7.5 shows the ECDF for the worst case relative rank for all three corpora. These graphs show the percentage of test sets for which a certain relative rank could be achieved.

These graphs show that Grouping significantly improves the ranking produced by Sfl. As the Info1 corpus contains the highest proportion of copied cells, Grouping shows the most effect on these test sets. Pre-Process Grouping performs poorly, in some cases even worse than the worst case scenario by Sfl alone, which can be seen in Figure 7.5 in the Info1 plot for the first 25 % of the relative rank.

Blocking shows an improvement for the Info1 corpus, but performs worse than Sfl for the Burnett corpus. This can be explained by the existence of oracle mistakes in this corpus. Dynamic Slicing shows almost no improvement on any of the three corpora.

Interestingly, Grouping applied to the union of negative Cones outperforms Sfl easily and nearly matches the results produced by Sfl with Grouping. Figure 7.6 shows the ECDF for the ranks produced for the union as well as Sfl on the Info1 corpus. While the same color legend is used for both approaches, the colors used to display the Sfl results are faded, as this is the same information as in Figure 7.5.

Let us inspect these results in more detail by using the *Impact* metric to measure the difference between the average rank with and without the strategies applied. Table 7.2 shows which percentage of improvable spreadsheets were actually impacted by the applied strategies. Post- and Combined Grouping perform best, improving the results for around 50 % of the spreadsheets in the Euses corpus and even more in the Info1 corpus, which contains the highest number of groupable cells. Due to the structure of the spreadsheets, Grouping has no impact on the results for the Burnett corpus at all. Blocking and Dynamic Slicing show considerably less positive impact, while exhibiting a higher negative impact.

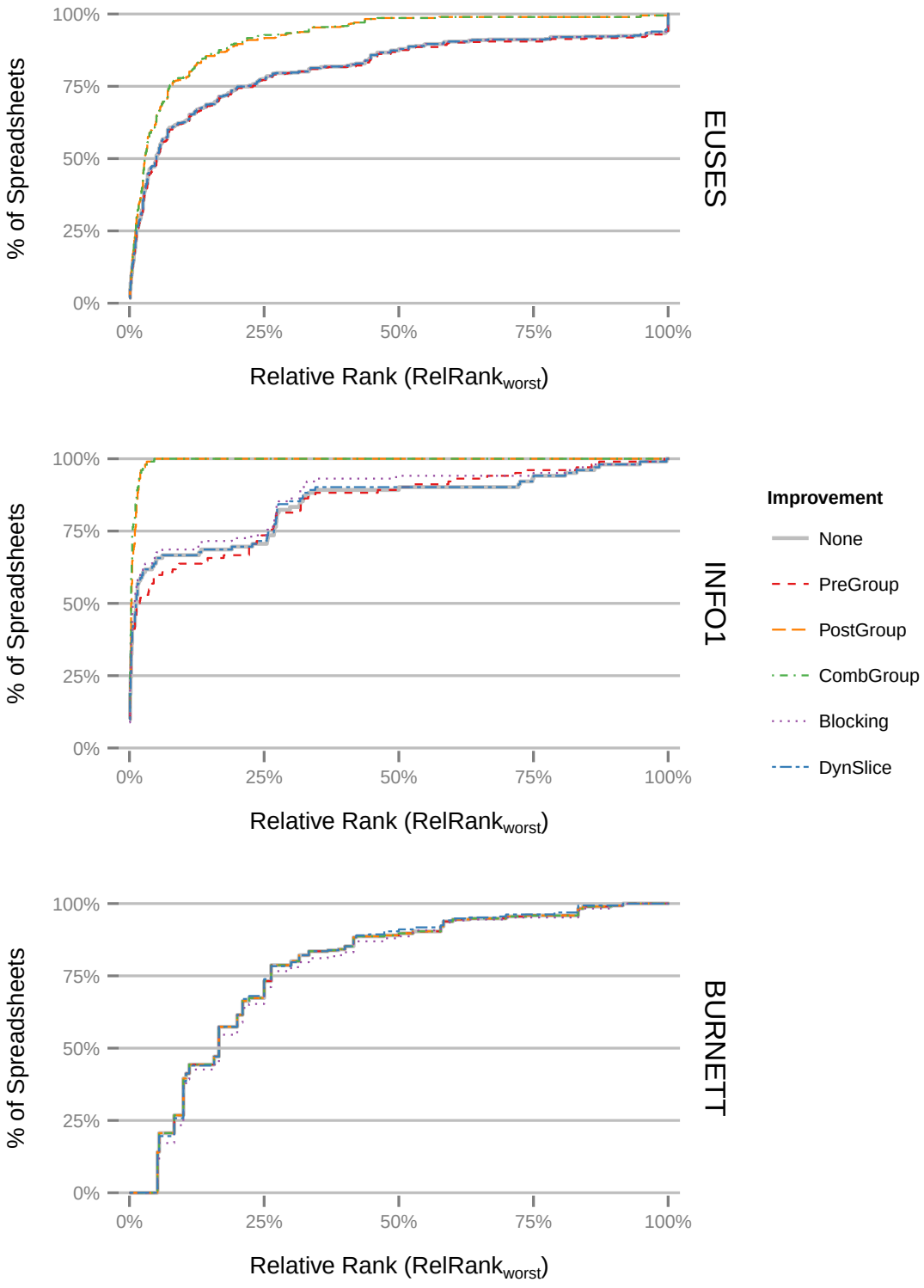


Figure 7.5: ECDF comparing relative rank of Grouping, Blocking and Dynamic Slicing achieved by SFL using the Ochiai coefficient.

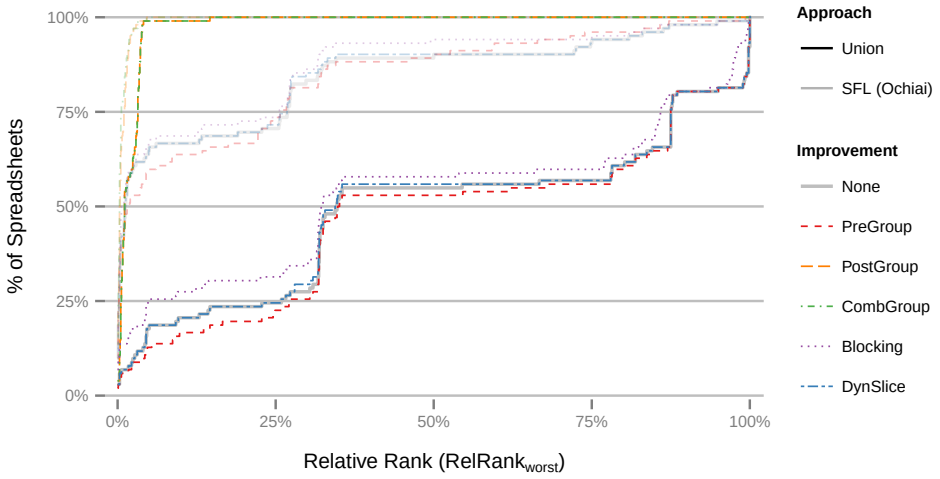


Figure 7.6: ECDF showing the relative rank produced by Grouping, Blocking and Dynamic Slicing applied to the union of negative Cones for the Info1 corpus. For comparison, the ranks produced by Sfl are displayed in faded colors.

Table 7.2: Percentage of test sets affected by the strategies Grouping, Blocking and Dynamic Slicing. We differentiate between positive, negative and no impact.

Corpus	*Impact*	PreGroup	PostGroup	CombGroup	Blocking	DynSlice
Euses	Positive	4	49.6	49.9	0	1.2
	Negative	2.4	0	0.9	0	0
	None	93.6	50.4	49.2	100	98.8
Info1	Positive	33	50	72.7	25	6.8
	Negative	22.7	0	2.3	3.4	0
	None	44.3	50	25	71.6	93.2
Burnett	Positive	0	0	0	6.2	9.7
	Negative	0	0	0	5.7	2.3
	None	100	100	100	88.1	88.1

Table 7.3 shows the distribution of the *Impact* the applied strategies achieved. No impact indicates no improvement over the average relative rank - meaning that either no ties were broken or they were broken in a manner that results in the faulty cell being ranked in the middle of the tie. If an approach reaches an *Impact* of 50 %, it means that the strategy successfully managed to rank the faulty cell at first position within the tie. As these approaches are not limited to improving the critical tie, it is also possible for them to reach higher impact values, such as CombGroup, which reaches a maximum positive *Impact* of 98 % for the Info1 corpus. This means that Grouping does not only succeed in breaking ties, it may even improve the best case absolute rank by reducing any ties that are ranked higher than the faulty cell. For the corpora without oracle mistakes, the strategies PostGroup, Blocking and DynSlice are safe and do not cause a negative impact. For the Burnett corpus, which contains a considerable number of oracle mistakes, both Blocking and DynSlice cause a negative effect. As these approaches remove cells or change their ranking, they are susceptible to any mistake the user makes when setting testing decisions. In the following paragraphs, we will discuss the findings for each approach individually.

Table 7.3: *Impact* of the strategies on the average case relative rank. An improvement of 50 % indicates that the strategy reaches the best case relative rank.

Corpus		PreGroup	PostGroup	CombGroup	Blocking	DynSlice
Euses	Min	-48.08	0	-0.33	0	0
	Q1	0	0	0	0	0
	Median	0	0	0	0	0
	Q3	0	9.52	9.52	0	0
	Max	27.4	53.9	53.9	0	2.66
	Average	-0.31	7.39	7.57	0	0.02
Info1	Min	-25.75	0	-0.04	-1.03	0
	Q1	0	0	0	0	0
	Median	0	0.72	1.22	0	0
	Q3	0.11	14.75	14.66	0.03	0
	Max	17.59	98.07	98.35	74.73	22.67
	Average	0.18	12.99	13.09	3.42	0.31
Burnett	Min	0	0	0	-36.84	-19.44
	Q1	0	0	0	0	0
	Median	0	0	0	0	0
	Q3	0	0	0	0	0
	Max	0	0	0	31.58	23.68
	Average	0	0	0	-0.24	0.37

Grouping

Grouping shows the most significant success of all compared strategies for Euses and Info1 test sets. Overall, the combined Grouping approach using both pre- and post-processing, is the most successful, yet only by a small margin compared to Post-Process Grouping. The only disadvantage of Post-Process Grouping is that it can only be applied to larger spreadsheets which use many copied formulas. PostGroup has no negative impact on the tested corpora even if the strategy could not be applied at all.

In contrast, Pre-Process Grouping shows little or even negative impact, in some cases performing worse than Sfl without any improvement strategies. This effect is most pronounced in the Info1 corpus and can be attributed to the fact that Pre-Process Grouping adds additional cells to the ranking which were not considered before. Pre-Process Grouping impacts only a small number of spreadsheets from the Euses corpus, which is likely due to the testing decisions being located in single result cells. It may perform better if the testing decisions were placed in formula cells which are part of a larger group.

No Grouping strategy could be applied to the Burnett corpus, as the spreadsheet in this corpus do not contain any copied formulas. This means that the influence of oracle mistakes on Grouping could not be evaluated. It is safe to assume that PostGroup would not aggravate any negative affects oracle mistakes have on Sfl, as the original ranking remains unchanged. This is due to the fact that no real reduction takes place with PostGroup, as the number of cells stays the same. We are only displaying critically tied statements with a more meaningful representation. For Pre-Process Grouping, which does change the number of cells in the ranking, it is unclear how oracle mistakes impede its effectiveness.

Blocking

Blocking is the second strategy which shows promise. The potential benefit of this approach can be seen by the positive impact on the Info1 and Burnett test sets. Due to the limitations of this approach, Blocking shows no effect if testing decisions are set in result cells only, as is the case for Euses. This strategy exhibits a negative impact on the fault localization result for the Burnett corpus, which can be explained by taking into account that the Burnett corpus contains oracle mistakes which impede the Blocking result. If Blocking is applied to the union of negative Cones for this corpus (rather than Sfl), it shows considerable improvement and partially simulates the results produced by Sfl.

Dynamic Slicing

Finally, Dynamic Slicing shows little to no improvement over the relative worst case rank. While this was to be expected for Euses due to the low number of spreadsheets containing `IF`s, both Info1 and Burnett contain many conditionals, yet show only slight improvements. While it is a low-risk strategy, negative impact may occur when testing decisions contain oracle mistakes. Please note that the implemented prototype

implements DynCones only for `IF` constructs. It is possible that the support of additional conditional structures such as the `LOOKUP` functions would allow the strategy to yield better results.

7.2.3 Tie-Breaking Strategies

In this section, we analyze how well the tie-breaking strategies perform, following a similar approach to the previous section. The strategies discussed in this section are: Cell Order Strategy (COS), Cell Distance Strategy (CDS) and Path Length Strategy (PLS) for the position-based tie-breaking strategies and Number of Operators (OP), Number of References (REF), Dispersion of References (DR), Cone Size (CS) and Cone Level (CL) for the metric-based tie-breaking strategies.

Metric-based strategies are based on the idea that some heuristics, such as a large number of operators in a formula, indicate a higher fault likelihood: The cell is less maintainable and the lack of structure in spreadsheet programming can easily lead to errors. Testing the suitability of these metrics for real-world examples is difficult, as none of our corpora contains entirely authentic test sets. Before analyzing the evaluation results, let us briefly discuss the attributes of each corpus.

Euses features faults that were injected in a random cell, therefore the results for the metric-based strategies on this corpus are not conclusive, as the fault did not originate from an end-user. Additionally, the testing decisions are set automatically for all result cells. This leads to a large number of testing decisions on one hand and possibly larger distances between the faulty cell and the testing decision on the other hand. This implies that CDS and PLS are not well suited for this corpus either.

The faults in the Burnett corpus were also injected, based on observed authentic faults. The high number of faults for such a small number of formula cells will likely skew the results for both distance- and metric-based tie-breaking strategies.

The Info1 corpus has very little variance, featuring similar structures such as groups in both base spreadsheets. The faults occurring in the test sets are, however, authentic, i.e. caused by the end-user. Additionally, the testing decisions, while also created automatically, are chosen from a mix of result and formula cells.

Of the three corpora, Info1 is best suited to evaluate tie-breaking. We will therefore focus especially on the results produced for the Info1 test sets in the following discussion.

Let us inspect the worst case relative ranks produced by Sfl with tie-breaking strategies. Figure 7.7 shows the worst case relative rank for the position-based tie-breaking strategies. The results are quite similar for all three approaches, with COS performing best for Euses and Burnett and worst for the Info1 corpus. For the Info1 corpus, CDS is the best performing approach of the three, albeit closely followed by PLS.

When we apply these strategies to the union rather than Sfl, we find once more that for some corpora, the primitive union with one of our proposed improvements performs better than Sfl even with the same improvement. This is the case for Figure 7.8, which shows that both CDS and the slightly lower PLS applied to the union clearly outperform the same strategies applied to Sfl. This may be an indicator that these strategies are especially suited to prioritize the authentic faults located in this corpus. The same results

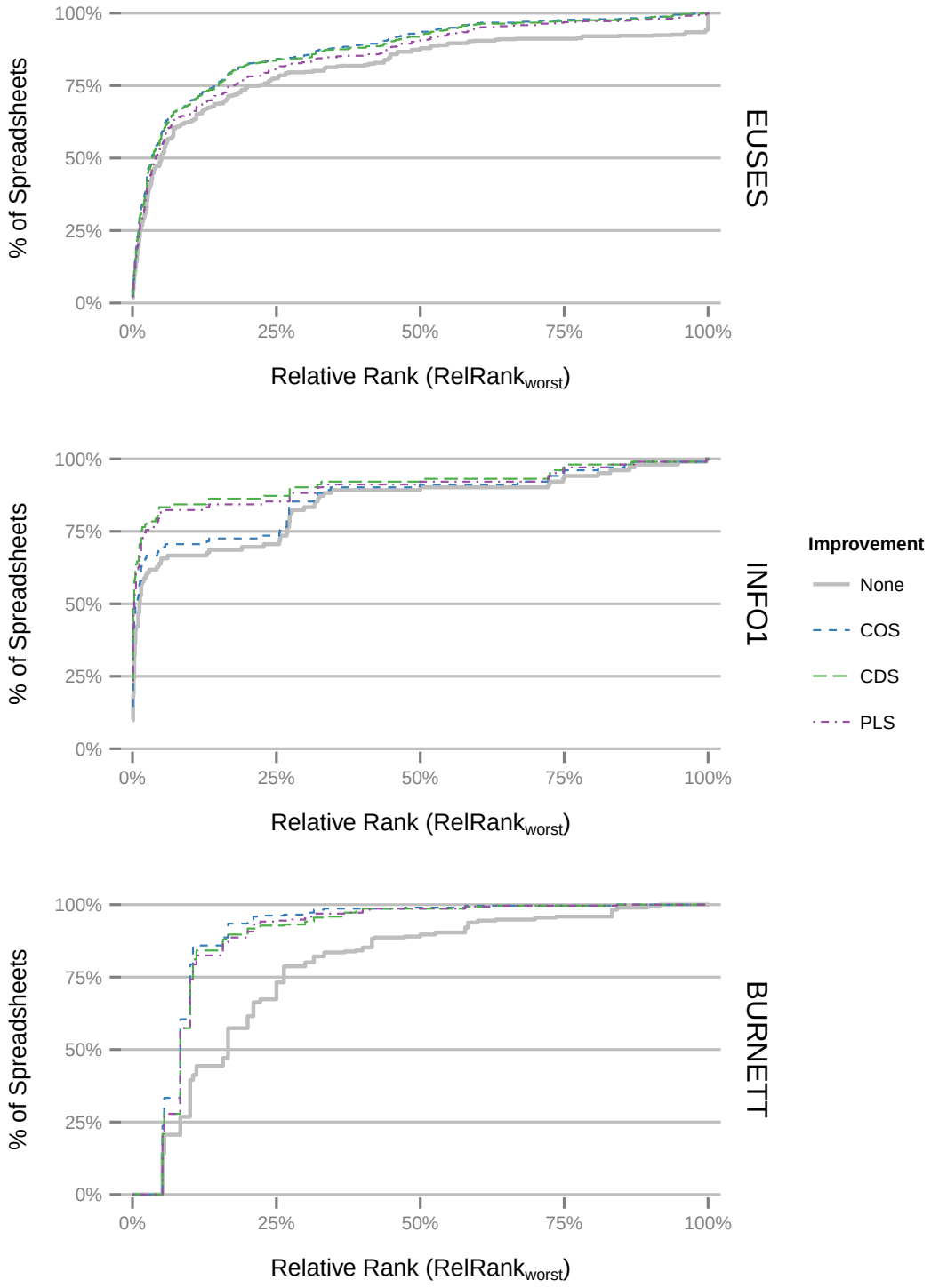


Figure 7.7: ECDF comparing relative rank of the position-based tie-breaking strategies Cell Order Strategy (COS), Cell Distance Strategy (CDS) and Path Length Strategy (PLS).

do not apply to the other corpora, where no such clear superiority exists. However, COS, which performs sub par on the Info1 corpus, is the best performing strategy for Burnett, again outperforming the same strategy applied to Sfl.

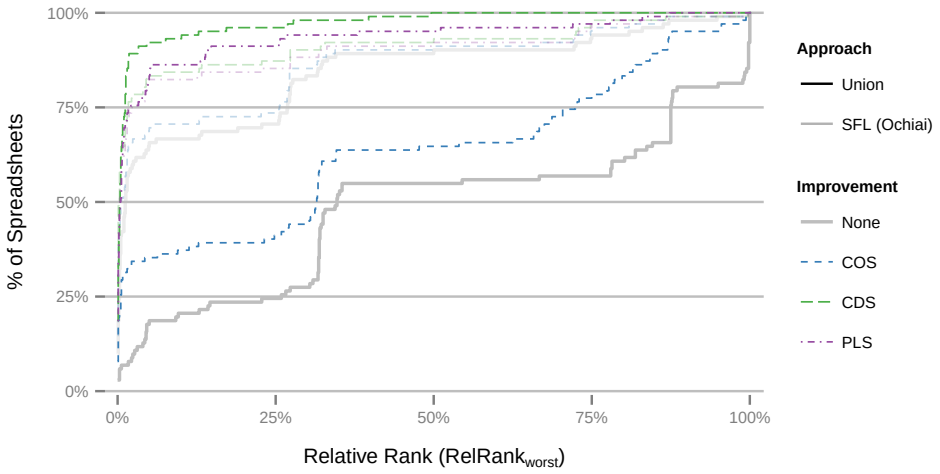


Figure 7.8: ECDF showing the relative rank for the union of negative Cones with applied position-based tie-breaking strategies for the Info1 corpus. For comparison, the results produced by Sfl are provided in the same but faded colors. CDS applied to the union clearly performs better than any other approaches, even CDS for Sfl.

The ECDF plots for metric-based approaches, seen in Figure 7.9, are not as conclusive. They show the CL and CS metrics performing quite well for Euses and Info1. For the Burnett corpus, REF closely followed by OP take the lead. DR shows only small improvements. In comparison, the results for the union, as seen in Figure 7.10 for the Info1 corpus, show a clearer superiority for CL, followed by CS. The other three approaches are all successful at significantly improving the rank produced by the union alone, whereas for Sfl, no approach has such a strong impact.

Additionally to these figures, Table 7.4 shows which portion of the test sets were affected either positively, negatively or not at all by the various strategies. The distribution of the *Impact* of the various strategies on each corpus can be seen in Table 7.5, with varied results and no clearly superior strategy.

For a more intuitive understanding of these results, Figure 7.11 shows box plots with removed outliers for the *Impact* produced by these approaches. Note that the three plots have considerably differing y-axis scales, with Info1 displaying all plots between -0.5 and 2 % and Burnett scaling between -10 and 20 %. One explanation for the high impact numbers for Burnett test sets is the size of the spreadsheets and the number of faults contained. It is more likely for a metric to rank one of many faulty cells at the highest position by chance if there exist only few formula cells. Tie-breaking has little impact on the results Sfl produced for Info1. It also often shows a negative impact for Euses test sets, which contain large spreadsheets and only few faults. It is notable that CDS shows a considerably higher positive impact than all other approaches for the

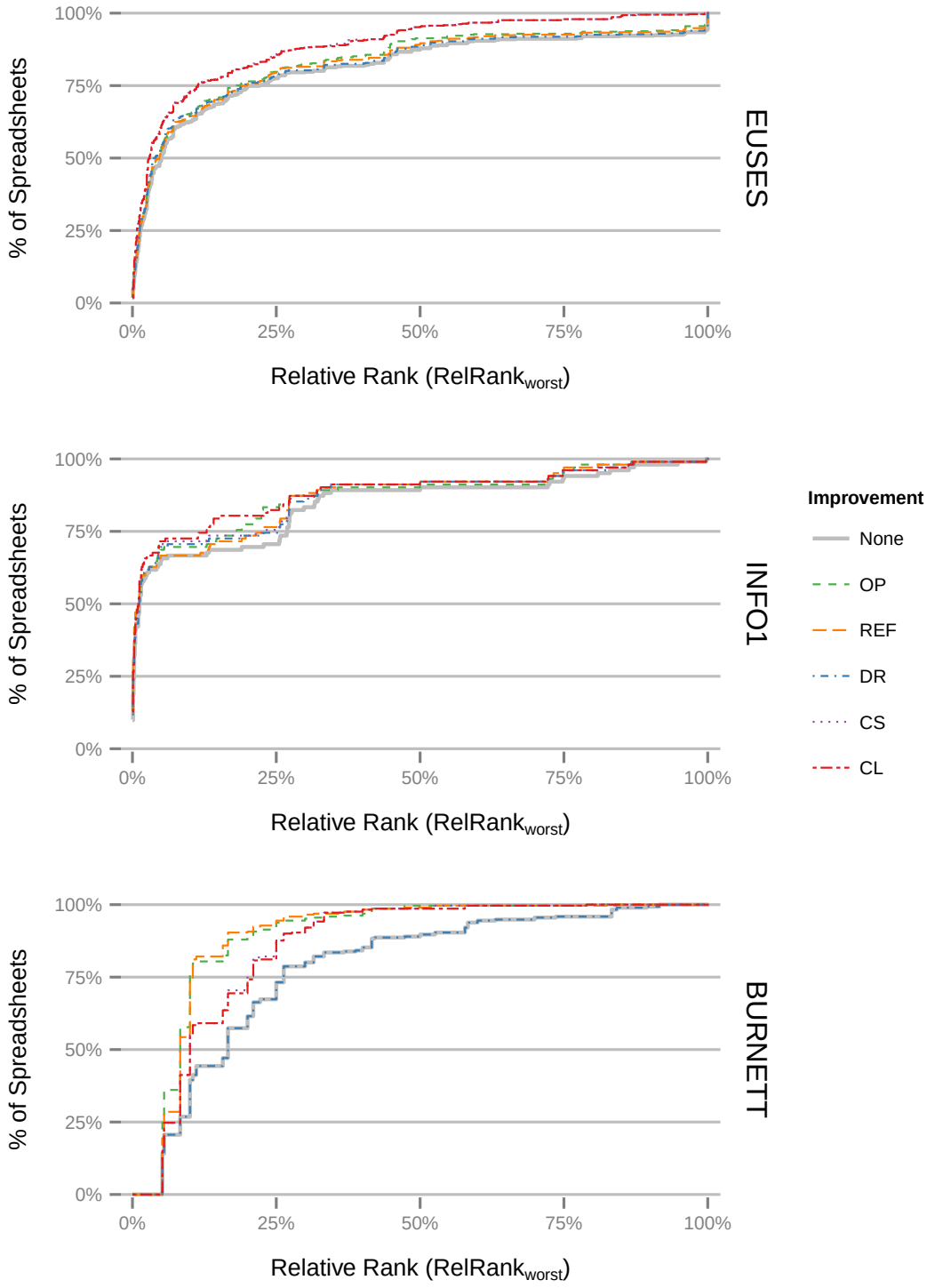


Figure 7.9: ECDF comparing relative rank of the approaches for the metric-based tie-breaking strategies Number of Operators (OP), Number of References (REF), Dispersion of References (DR), Cone Size (CS) and Cone Level (CL).

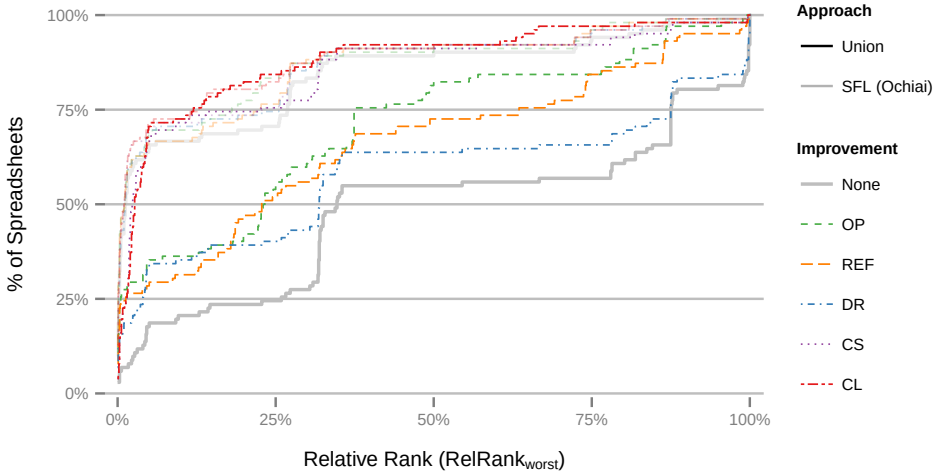


Figure 7.10: ECDF showing the relative rank for metric-based tie-breaking strategies applied to the union of negative CONEs for the INFO1 corpus. These values are compared to SFL, which are also shown in identical but faded colors. CL applied to the union performs similarly well as the same metric applied to SFL.

Table 7.4: Percentage of test sets affected by the tie-breaking strategies Cell Order Strategy (COS), Cell Distance Strategy (CDS) and Path Length Strategy (PLS) for the position-based and Number of Operators (OP), Number of References (REF), Dispersion of References (DR), Cone Size (CS) and Cone Level (CL) for the metric-based tie-breaking strategies. We differentiate between positive, negative and no impact.

		Position-based			Metric-based				
Corpus	*Impact*	COS	CDS	PLS	OP	REF	DR	CS	CL
EUSES	Positive	35.8	31.1	20.7	25.6	17.6	10.4	65.4	63.8
	Negative	60	63.8	62.1	26.6	48.7	15.5	21.9	21.4
	None	4.2	5.2	17.2	47.8	33.6	74.1	12.7	14.8
INFO1	Positive	43.2	62.5	54.5	37.5	40.9	13.6	35.2	42
	Negative	39.8	12.5	22.7	40.9	33	30.7	43.2	34.1
	None	17	25	22.7	21.6	26.1	55.7	21.6	23.9
BURNETT	Positive	87.5	78.4	78.4	83.5	88.1	0	56.8	56.2
	Negative	8	4.5	3.4	1.1	7.4	0	35.8	37.5
	None	4.5	17	18.2	15.3	4.5	100	7.4	6.2

Table 7.5: *Impact* of the tie-breaking strategies on the average case relative rank. The maximum impact is 50 %, indicating that the strategy matches the previous best case scenario.

		Position-based			Metric-based				
Corpus		COS	CDS	PLS	OP	REF	DR	CS	CL
Euses	Min	-49.11	-49.11	-49.11	-46.67	-49.63	-18.75	-45.9	-45.9
	Q1	-4.23	-4.88	-1.67	-0.07	-1.52	0	0	0
	Median	-0.42	-0.45	-0.34	0	0	0	0.34	0.34
	Q3	0.7	0.63	0	0.1	0	0	1.77	1.67
	Max	45.9	45.9	45.9	41.67	47.5	33.33	49.11	49.11
	Average	-2.43	-2.95	-2.23	0.26	-2.12	0.11	2.26	2.26
Info1	Min	-40.47	-0.17	-23.42	-33.37	-36.92	-3.64	-40.44	-40.44
	Q1	-0.14	0	0	-0.09	-0.09	-0.02	-0.15	-0.09
	Median	0	0.07	0.02	0	0	0	0	0
	Q3	0.09	0.55	0.2	0.08	0.09	0	0.05	0.08
	Max	41.71	47.31	45.07	15.83	36.06	45.11	40.45	44.33
	Average	-1.69	3.88	2.92	-0.63	-0.69	0.95	-0.91	0.27
Burnett	Min	-5.56	-16.67	-5	-5.56	-7.89	0	-10.53	-10.53
	Q1	2.78	4.17	4.17	2.78	4.17	0	-5	-4.17
	Median	5	5.26	5.26	5.26	5.26	0	2.78	2.78
	Q3	10.53	10.53	10.53	10.13	10.53	0	8.33	8.33
	Max	42.11	37.5	37.5	39.47	42.11	0	29.17	29.17
	Average	8.84	8.09	8.52	8.39	8.36	0	3.45	3.59

INFO1 test sets.

Finally, Table 7.6 shows how well the tie-breaking strategies work at splitting up the critical ties, regardless of their success at ranking the faulty cell higher than non-faulty cells. COS is best at breaking up ties, and REF, unsurprisingly, is worst. A low *Tie-Reduction* value is not necessarily an indication of poor performance: A strategy with potentially high negative *Impact* and high *Tie-Reduction* leads to worse results than a strategy with low *Impact* and low *Tie-Reduction*. Focusing on the INFO1 corpus, let us now discuss these results for each approach in more detail.

Table 7.6: *Tie-Reduction* of the strategies on the average case relative rank, showing how well the strategies break up the critical ties, regardless of the ranking of the faulty cell.

		Position-based			Metric-based				
Corpus		COS	CDS	PLS	OP	REF	DR	CS	CL
Overall Average		72	65.2	50.7	47.1	48.4	10	56.8	53.7
EUSES	Min	0	0	0	0	0	0	0	0
	Q1	75	75	20	0	0	0	20	20
	Median	85.7	85.7	50	12.2	33.3	0	50	50
	Q3	97	96.7	80	50	60	1.6	83.3	80
	Max	100	100	99.7	99.5	100	99.7	99.7	99.7
	Average	82.2	80.3	48.9	23.4	36.1	10.8	51.8	50.8
INFO1	Min	0	0	0	0	0	0	0	0
	Q1	50	31.2	20	50	33.3	0	45.5	25
	Median	80	66.7	50	66.7	58.7	0	64.6	50
	Q3	94.9	89.7	68.8	83.3	66.7	29.7	92.4	87.2
	Max	100	100	100	99.9	99.8	99.9	100	100
	Average	67.5	58.9	49.5	60	49.9	19.1	59	53.8
BURNETT	Min	0	0	0	0	0	0	0	0
	Q1	50	50	50	50	50	0	50	50
	Median	66.7	63.3	60	66.7	50	0	60	57.1
	Q3	80	75	75	75	71.4	0	66.7	66.7
	Max	94.1	90.9	90.9	93.8	94.1	0	88.9	88.9
	Average	66.3	56.4	53.7	58	59.1	0	59.5	56.5

Cell Order Strategy (COS)

The cell order strategy breaks the ties the most consistently, cutting the size of the critical tie in half or better for at least 75 % of test sets. It performs best of all tie-breaking strategies for the BURNETT corpus, but has a negative average impact for the other two corpora, indicating a high risk. For the BURNETT corpus, the success may also be due to the structure of the spreadsheets, as it features small spreadsheets with

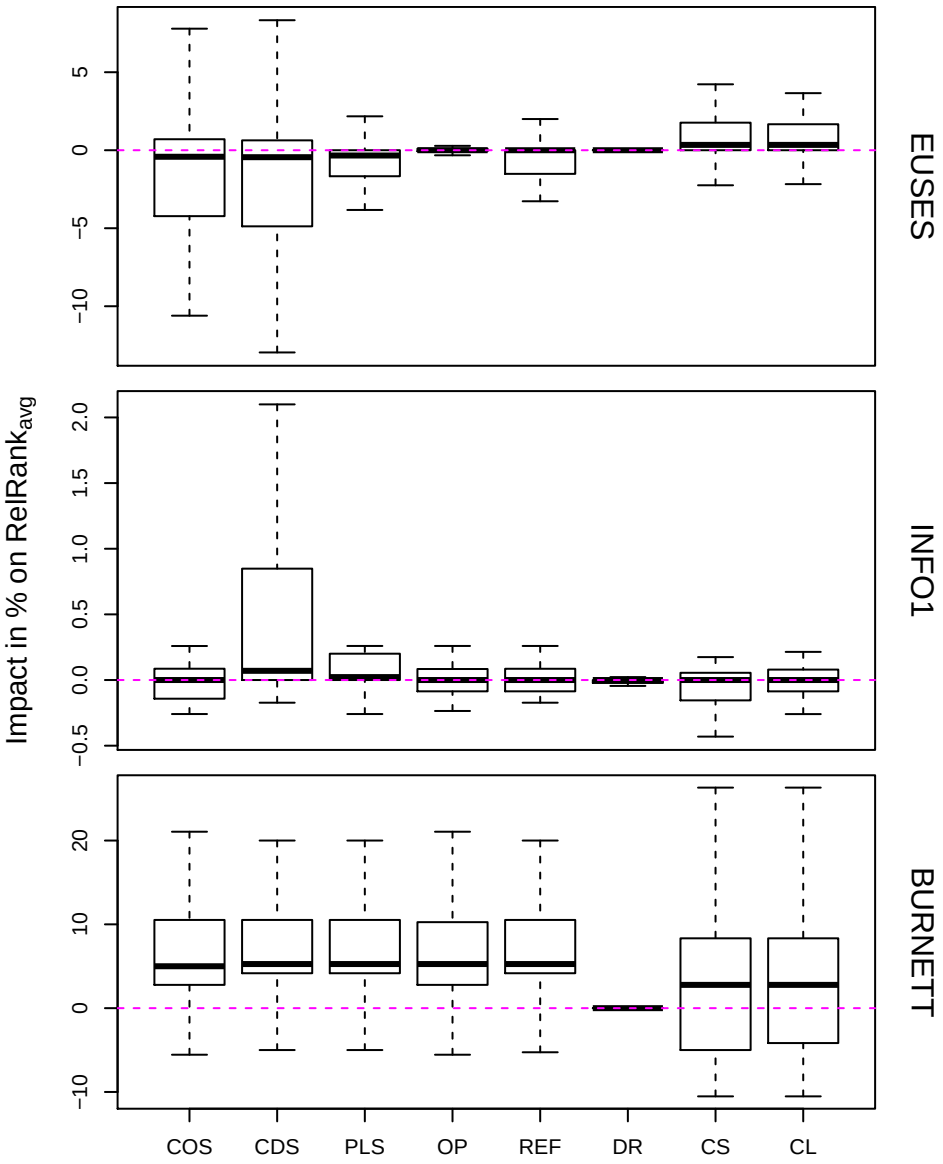


Figure 7.11: Box plots comparing the *Impact* of tie-breaking on EUSES, INFO1 and BURNETT (from top to bottom).

multiple faults. This hypothesis is supported by the fact that COS applied to the union of negative cones performs better than SFL with or without any improvement strategies.

Cell Distance Strategy (CDS)

CDS performs best for all tie-breaking strategies in the INFO1 corpus, with the smallest negative impact and an average positive impact of 4 %. The median shows that this metric can improve the average case scenario for more than 50 % of the spreadsheets. For INFO1, similarly to COS in BURNETT, CDS applied to the union of negative CONEs, again surpasses the rankings received for SFL.

For the EUSES corpus, all position-based strategies show a negative impact. One explanation for this discrepancy is the location of the testing decisions: CDS measures the distance between a cell and the nearest negative testing decision it participates in. Both INFO1 and BURNETT provide testing decisions for any formula cells, whereas EUSES only has testing decisions for result cells. It is likely that result cells are on the edges of the spreadsheet which may result in larger distances from faults.

Path Length Strategy (PLS)

PLS performs similarly to CDS, though with considerable more risk for the INFO1 corpus and less risk for the BURNETT corpus. It has a negative average impact on the EUSES corpus. As we suspected, this strategy is less successful at *Tie-Reduction* than its counterpart CDS.

Number of Operators (OP)

OP can moderately improve results for EUSES and INFO1 but also exhibits high risks for these corpora. The positive result achieved for BURNETT is, again, likely due to the high number of faults in the spreadsheet, leading to many ties being broken by chance.

Number of References (REF)

REF performs similarly to OP, moderately improving results for INFO1 and EUSES. However, as opposed to OP, it has a higher negative average *Impact* of -2.1 % on the EUSES test set. In general, REF affects many test sets negatively for this corpus, with almost 50 % negatively impacted and only 17 % positively affected. It is likely that the success for the BURNETT corpus is due to similar reasons as for OP.

Dispersion of References (DR)

The DR metric targets a specific property in formula cells and therefore requires specific conditions to be met for this approach to be successful. For the BURNETT corpus, this strategy could not be applied at all, as all cell references used in these spreadsheets are absolute due to the use of named cells. However, for EUSES and INFO1, DR shows a slight positive average impact and comes with hardly any risk of negative impact. Between

all metric-based strategies in Info1, DR has the highest average, as it poses the least threat of negative impact while also potentially allowing a high positive impact.

Cone Size and Cone Level (CS and CL)

Finally, the metrics based on Cone size and levels perform quite similarly. The outperform all other tie-breaking strategies on the Euses corpus, with the best *Impact* values, but also pose the highest risk. This high risk is emphasized by their low performance for the Info1 corpus, where, on average, CS has a negative impact and CL has a low positive impact. For the Burnett corpus, the strategies have a positive average impact, yet a considerably lower one than any other tie-breaking strategy. Their impact values are also distinct from the other approaches in Figure 7.11, which shows that CL and CS have a high negative impact for many test sets of the Burnett corpus.

In this evaluation, we learned that Post-Process and Combined Grouping show great potential to reduce the user's search space with little to no risk. Input reduction strategies can only moderately improve results and the success of these strategies is susceptible to oracle mistakes. Tie-breaking strategies are more difficult to evaluate as they require diverse and authentic test sets. We have established that for authentic faults, the Cell Distance Strategy (CDS) has the lowest risk while providing high *Tie-Reduction* and an overall positive *Impact*. Metric-based strategies tend to have little impact, however, the low-risk Dispersion of References (DR) strategy shows some potential. While DR can only be applied in specific cases and therefore has a low *Tie-Reduction* value, in cases where it *can* be applied, it is possible to achieve high positive *Impact* values.

7.2.4 Runtimes

To conclude the evaluation, let us briefly discuss which runtimes to expect from our approaches. While runtime performance was not the focus of this work, the following tables will show the average runtimes for all test sets, using the average value of ten executions for each test set. The runtimes for Euses are the most representative, due to its large size and diversity of spreadsheet sizes. Info1 has a higher average of formula cells and therefore provides examples for larger spreadsheets. The results for the Burnett corpus are negligible, as the corpus only contains two small spreadsheets.

Table 7.7 shows the runtimes for Grouping, Blocking and Dynamic Slicing. Overall, PostGroup performs best with an average of 9.9 ms, followed by DynSlice. Blocking is the most expensive approach for the Info1 corpus. However, its runtime for Euses is not meaningful, as the strategy could not be applied to this corpus at all. CombGroup has the longest runtime for Euses, although clearly the PreGroup part of the combination is more expensive than PostGroup.

Table 7.8 lists the runtimes for the tie-breaking strategies, which we expected to be less expensive than approaches from the other category. The table shows, however, that for the Info1 corpus, CS and CL are by far the most computationally complex

Table 7.7: Average runtime in milliseconds for SFL with Ochiai and the improvements Grouping, Blocking and Dynamic Slicing.

Corpus	None	PREGROUP	POSTGROUP	COMBGROUP	BLOCKING	DYNSLICE
EUSES	0.2	60.8	1.4	61.9	24	10.9
INFO1	1.7	52	9.9	60.9	64.1	29.9
BURNETT	3e-02	0.2	0.1	0.2	0.3	0.2

approaches, reaching the average of 186 ms and 176.2 ms respectively. These numbers indicate that there is need for improvement for the CONE function implemented in the prototype, which is called for every cell in the ranking to compute the metric. The fastest tie-breaking strategy is OP, followed by PLS, performing almost three times faster than the other strategies for the INFO1 test sets.

Table 7.8: Average runtime in milliseconds for SFL with Ochiai and the tie-breaking strategies.

		Position-based			Metric-based				
Corpus	None	COS	CDS	PLS	OP	REF	DR	CS	CL
EUSES	0.2	0.5	0.7	0.6	0.4	2.4	2.4	5.3	4.8
INFO1	1.7	13.4	12.7	4	3.2	11.5	11.8	186	176.2
BURNETT	3e-02	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Even high numbers such as 200 ms are an acceptable runtime in comparison to approaches other than SFL. As an example, model-based fault localization approaches [5], using constraint-solver back-ends, will likely take significantly longer than any of the improvement strategies discussed in this thesis.

8 Conclusion

Many critical computations depend on the correctness of spreadsheets, yet studies estimate that most in-production spreadsheets contain errors. Spectrum-based fault localization is one of many possible approaches to improve the quality of existing spreadsheets. To ensure that the developed approaches are helpful to end-users, we must provide efficient and unambiguous feedback, for example in the form of ranked cells.

Our work provides important progress in this direction, improving fault localization techniques, in particular SFL, to return a more effective ranking. We developed Grouping and adapted input reduction approaches and tie-breaking strategies for spreadsheets. To evaluate these improvements, we introduced two new spreadsheet corpora, one of which contains authentic faults, the other authentic testing decisions including oracle mistakes. These new corpora are valuable to future research, as they simulate user interaction and give an insight into the structural requirements of various fault localization techniques. Note that while we describe and evaluate the improvements on the basis of the ranking returned by SFL, nearly all of our contributions can easily be applied to other trace- and model-based fault localization techniques.

The key finding of this thesis is the success of Post-Process and Combined Grouping. Spreadsheets often contain many copied formulas and data, but this property was not exploited by fault localization techniques so far. We have found that the size of ties can be reduced drastically with the Post-Process Grouping approach, leading to significantly better rankings. This approach comes with little to no disadvantages: It produces only little overhead in addition to SFL and exhibits no risk of negatively impacting the results.

In contrast, Pre-Process Grouping alone does not appear to be an effective improvement, as it tends to increase the number of cells in the ranking due to copied testing decisions. In combination with Post-Process Grouping, this issue can be alleviated, but Combined Grouping does not show a significant difference to Post-Process Grouping alone. The low impact rate also indicates that this strategy is applicable for only a small subset of test sets, requiring specific conditional requirements.

The input reduction strategies proposed in this thesis moderately increase effectiveness, strongly depending on the structural properties of the spreadsheet and the testing decisions. Both Blocking and Dynamic Slicing increase runtime significantly and show less impact than grouping or tie-breaking strategies. As they depend on correct testing decisions, which cannot be guaranteed in practice, we do not recommend these improvements for novice users. However, more experienced spreadsheet users that are familiar with the structures of the spreadsheet under inspection and the risk exhibited by these strategies could be given the choice to enable the approaches on a case-by-case basis.

An important point in this thesis is to provide the ground-work for future research in the topic of tie-breaking, creating suitable test data and providing a small selection

of implemented metrics. Tie-breaking strategies are vital when presenting the user with the fault localization result, as large ties may frustrate the user and the faulty cell may be found only after inspecting all other, non-faulty cells in the tie. The evaluation shows that tie-breaking strategies—while also highly dependent on spreadsheet structure and accompanied by a higher risk than Grouping—show great promise to reduce the issue of lacking prioritization. We found the tie-breaking strategy CDS, which uses a cell's distance from negative testing decisions to increase its fault likelihood, to perform best, assuming that testing decisions are not exclusively set for result cells.

The results of our evaluation are not always conclusive, as each strategy affects the uniquely structured testing corpora differently. Burnett is the only corpus where testing decisions contain oracle mistakes, which allows us to evaluate how robust our proposed techniques are against imperfect user input. Due to the small size of the spreadsheets and the corpus itself, the values computed for the tie-breaking strategies are less meaningful. The Grouping approach could not be applied to the test sets in Burnett at all, as the spreadsheets do not contain copied formulas. Euses has the greatest variance of spreadsheets, but approaches such as Blocking and some tie-breaking strategies show no effect on its test sets due to the artificial injection of faults and testing decisions. Of the three considered corpora, Info1 is the only corpus with authentic faults, but still has artificial testing decisions and is based on only two spreadsheets, which both feature copied formulas heavily.

There is still need to extend the existing test data. Large spreadsheet corpora with authentic faults are needed to evaluate metric-based improvements in more detail. One of our goals is to make the corpora used in this work publicly available, allowing other researchers access to the same data and facilitating further research, discussion and results for comparison.

There are many additional metrics and code-smells [7, 12] which can be implemented and evaluated in addition to the ones we mentioned. Some of the metrics used in this thesis may also be inverted. For example, the Cone size and Cone level metrics perform poorly for the Burnett corpus and may produce better results with inverted definitions. Additionally, it would be interesting to evaluate how the discussed approaches perform when combined with one another, possibly producing synergy effects that have not been investigated in this thesis.

Grouping is shown to have great potential in improving fault localization, yet, especially Pre-Process Grouping, requires more examination and research to produce effective results. The current prototype may be improved, for example, by loosening the grouping criteria and allowing more cells to be collapsible. Moreover, the approach might be improved significantly by treating a group of cells as a single cell rather than copying testing decisions to each cell in the group, therefore reducing the number of cells participating in test cases. Another possible improvement is to allow a single cell to participate in more than one group and therefore allowing more grouping scenarios. Even in its current form, we are confident that the improvements produced by Grouping are a valuable addition to any spreadsheet debugging tool.

Bibliography

[1] R. Abraham and M. Erwig, "Goal-directed debugging of spreadsheets," in *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2005, pp. 37–44, ISBN: 0-7695-2443-5, DOI: 10.1109/VLHCC.2005.42.

[2] R. Abraham and M. Erwig, "Header and unit inference for spreadsheets through spatial analyses," in *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2004, pp. 165–172, ISBN: 0-7803-8696-5, DOI: 10.1109/VLHCC.2004.29.

[3] R. Abraham and M. Erwig, "Mutation operators for spreadsheets," *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 1, pp. 94–108, Jan. 2009, ISSN: 0098-5589, DOI: 10.1109/TSE.2008.73.

[4] R. Abreu, P. Zoeteweij, and A. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *12th Pacific Rim International Symposium on Dependable Computing, 2006. PRDC '06*, Dec. 2006, pp. 39–46, DOI: 10.1109/PRDC.2006.18.

[5] 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 *Proceedings of the 13th International Conference on Quality Software (QSIC)*, 2013, pp. 139–148, ISBN: 978-0-7695-5039-8, DOI: 10.1109/QSIC.2013.46.

[6] Y. Ayalew and R. Mittermeir, "Spreadsheet debugging," in *Proceedings of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2003, pp. 67–79, [Online]. Available: http://arxiv.org/abs/0801.4280 (visited on 04/09/2014).

[7] A. Bregar, "Complexity metrics for spreadsheet models," *Proceedings of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2004, [Online]. Available: http://arxiv.org/abs/0802.3895 (visited on 04/03/2014).

[8] T. Y. Chen and Y. Y. Cheung, "On program dicing," *Journal of Software Maintenance*, vol. 9, no. 1, pp. 33–46, Feb. 1997, ISSN: 1040-550X, DOI: 10.1002/(SICI)1096-908X(199701)9:1<33::AID-SMR143>3.3.CO;2-W.

[9] N. DiGiuseppe and J. Jones, "Fault interaction and its repercussions," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 3–12, DOI: 10.1109/ICSM.2011.6080767.

[10] M. Fisher and G. Rothermel, "The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms," in *Proceedings of the First Workshop on End-user Software Engineering (WEUSE)*, 2005, pp. 1–5, ISBN: 1-59593-131-7, DOI: 10.1145/1082983.1083242.

[11] E. Getzner, "Survey of fault localization techniques in spreadsheets," University of Technology Graz, Diploma Seminar, 2014.

[12] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2012, pp. 409–418, DOI: 10.1109/ICSM.2012.6405300.

[13] B. Hofer and F. Wotawa, "Mutation-based spreadsheet debugging," in *Proceedings of the 5th IEEE International Workshop on Program Debugging (IWPD), IEEE 24th International Symposium on Software Reliability Engineering (ISSRE) sup. mat.*, Nov. 2013, pp. 132–137, DOI: 10.1109/ISSREW.2013.6688892.

[14] B. Hofer, A. Perez, R. Abreu, and F. Wotawa, "On the empirical evaluation of similarity coefficients for spreadsheets fault localization," *Automated Software Engineering*, pp. 1–28, 2014, ISSN: 0928-8910, 1573-7535, DOI: 10.1007/s10515-014-0145-3.

[15] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner, "On the empirical evaluation of fault localization techniques for spreadsheets," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013, pp. 68–82, ISBN: 978-3-642-37056-4, DOI: 10.1007/978-3-642-37057-1_6.

[16] B. Hofer and F. Wotawa, "Spectrum enhanced dynamic slicing for better fault localization," in *European Conference on Artificial Intelligence (ECAI)*, 2012, pp. 420–425.

[17] W. Högerle, F. Steimann, and M. Frenkel, "More debugging in parallel," *25th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 133–143, 2014, [Online]. Available: http://www.fernuni-hagen.de/imperia/md/content/ps/moredebugginginparallel.pdf (visited on 01/26/2015).

[18] D. Jannach and U. Engler, "Toward model-based debugging of spreadsheet programs," in *Proceedings of the 9th Joint Conference on Knowledge-Based Software Engineering (JCKBSE)*, 2010, pp. 252–264, [Online]. Available: http://ls13-www.cs.tu-dortmund.de/homepage/publications/jannach/Conference_JCKBSE_2010.pdf (visited on 06/25/2014).

[19] 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*, vol. 94, pp. 129–150, Aug. 2014, ISSN: 0164-1212, DOI: 10.1016/j.jss.2014.03.058.

[20] J. Kwak. (Feb. 9, 2013). The importance of excel, The Baseline Scenario. @, [Online]. Available: http://baselinescenario.com/2013/02/09/the-importance-of-excel/ (visited on 03/31/2015).

[21] J. Lawrance, R. Abraham, M. Burnett, and M. Erwig, "Sharing reasoning about faults in spreadsheets: an empirical study," in *Proceedings of the 2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2006, pp. 35–42, ISBN: 0-7695-2586-5, DOI: 10.1109/VLHCC.2006.43.

[22] Y. Miao, Z. Chen, S. Li, Z. Zhao, and Y. Zhou, "Identifying coincidental correctness for fault localization by clustering test cases," in *SEKE*, 2012, pp. 267–272, [Online]. Available: http://software.nju.edu.cn/zychen/paper/2012SEKE1.pdf (visited on 01/26/2015).

[23] R. Mittermeir and M. Clermont, "Finding high-level structures in spreadsheet programs," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings*, 2002, pp. 221–232, DOI: 10.1109/WCRE.2002.1173080.

[24] R. R. Panko, "Spreadsheet errors: what we know. what we think we can do," in *Proceedings of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000, pp. 7–17, [Online]. Available: http://arxiv.org/abs/0802.3457 (visited on 04/08/2014).

[25] A. Podgurski and C. Yang, "Partition testing, stratified sampling, and cluster analysis," in *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 1993, pp. 169–181, ISBN: 0-89791-625-5, DOI: 10.1145/256428.167076.

[26] 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 (DSL)*, 1999, pp. 25–38, ISBN: 1-58113-255-7, DOI: 10.1145/331960.331968.

[27] J. R. Ruthruff, M. Burnett, and G. Rothermel, "Interactive fault localization techniques in a spreadsheet environment," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 4, pp. 213–239, Apr. 2006, ISSN: 0098-5589, DOI: 10.1109/TSE.2006.37.

[28] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, 1981, pp. 439–449, ISBN: 0-89791-146-6, [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557 (visited on 10/17/2014).

[29] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, vol. 1, Jul. 2007, pp. 449–456, ISBN: 0-7695-2870-8, DOI: 10.1109/COMPSAC.2007.109.

[30] F. Wotawa, "Fault localization based on dynamic slicing and hitting-set computation," in *2010 10th International Conference on Quality Software (QSIC)*, Jul. 2010, pp. 161–170, DOI: 10.1109/QSIC.2010.51.

[31] F. Wotawa, "Bridging the gap between slicing and model-based diagnosis.," in *SEKE*, 2008, pp. 836–841, [Online]. Available: http://www.researchgate.net/publication/221390019_Bridging_the_Gap_Between_Slicing_and_Model-based_Diagnosis/file/d912f50ed8da755e9f.pdf (visited on 07/30/2014).

[32] X. Xu, V. Debroy, W. Eric Wong, and D. Guo, "Ties within fault localization rankings: exposing and adressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 06, pp. 803–827, Sep. 2011, ISSN: 0218-1940, 1793-6403, DOI: 10.1142/S0218194011005505.