Elisabeth Getzner

Survey of Fault Localization Techniques for Spreadsheets

716.117 Diploma Seminar

Graz University of Technology

Institute for Software Technology

Supervisor: Univ.-Prof. Dipl-Ing. Dr.techn. Franz Wotawa,
Dipl.-Ing. Dr.techn. Birgit Hofer

Graz, October 2014

Abstract

Spreadsheets are widely used by end-users in the corporate environment, as they allow the display and calculation of large amounts of data. Real-life application spreadsheets are often the basis for critical computations, yet research has found that many of them contain errors, which could cause financial loss if left uncorrected.

In this paper, we examine fault localization techniques which support the end-user in locating such errors. We provide an overview and comparison of the research in this area, including trace-based, model-based and combined approaches. Among others, we discuss the techniques Wysiwyt, Sfl, Sendys as well as ConBug and Exquisite and explain their process to locate faults. These approaches are compared with regards to their user input, runtime complexity, their ability to handle multiple faults and their outputs. Additionally, we provide an overview of the evaluations for the presented tools, comparing their setup and the metrics used to measure the success of these approaches.

Contents

1 Introduction and Scope 1
1.1 Comparison Criteria . . . 2
1.2 Basic Definitions and Running Example . . . 3

2 Trace-based Approaches 6
2.1 Input . . . 6
2.2 Algorithms . . . 11
2.3 Output . . . 20
2.4 Fault Complexity . . . 23
2.5 Evaluations . . . 28
2.6 Comparison . . . 32

3 Model-based Approaches 37
3.1 Input . . . 38
3.2 Approach . . . 40
3.3 Output . . . 43
3.4 Evaluation . . . 45
3.5 Comparison . . . 48

4 Combined Approaches 50
4.1 Spectrum ENhanced DYnamic Slicing (SENDYS) . . . 50
4.2 UCheck and WYSIWYT . . . 55

5 Repair Approaches 58

6 Conclusion 60
6.1 User Input . . . 60
6.2 Algorithms . . . 61
6.3 Output . . . 62
6.4 Summary . . . 62

List of Figures

1.2.1 Single fault example . . . 3
1.2.2 Multiple fault example . . . 4
1.2.3 Partial Program Dependency Graph for Example 1.2.1 . . . 5
1.2.4 Partial Program Dependency Graph for Example 1.2.2 . . . 5

2.1.1 Testing decisions for Example 1.2.1 . . . 7
2.1.2 Test case visualization . . . 10
2.2.1 Hit-spectra matrix and error vector e . . . 17
2.2.2 Computation steps for MostInfluential . . . 19
2.3.1 Output visualization for Wysiwyt . . . 21
2.3.2 Output visualization for Sfl . . . 22
2.3.3 Output visualization for MostInfluential . . . 22
2.4.1 Differences between single and multiple faults . . . 23
2.4.2 Testing decisions for Example 1.2.2 (independent faults) . . . 24
2.4.3 Partial Program Dependency Graph for Example 1.2.2 . . . 25
2.4.4 Testing decision for Example 1.2.2 (dependent faults) . . . 26

3.2.1 The HS-DAG to compute the hitting sets for Example 1.2.2 . . . 41

4.1.1 Output visualization for Sfl . . . 53

List of Tables

2.1.1 Test case participation 9
2.2.1 Fault likelihood values for Test Count 12
2.2.2 Fault likelihood values for Blocking 14
2.2.3 Fault likelihood values for Nearest Consumer 16
2.2.4 Fault likelihood values for SFL using Ochiai 18
2.3.1 Fault likelihood levels for WYSIWYT 20
2.5.1 Effectiveness metric for trace-based approaches 29
2.5.2 Effectiveness metric for values greater than zero 30
2.5.3 Ranking metric for trace-based approaches 31
2.6.1 Comparison of all trace-based approaches 33
2.6.2 User input for trace-based approaches 34
2.6.3 Information bases for trace-based approaches 35
2.6.4 Evaluation comparison for WYSIWYT and SFL 35

3.5.1 Comparison of the model-based approaches 48
3.5.2 User input for model-based approaches 49

4.1.1 Fault likelihood values for SENDYS 52
4.1.2 Comparison between WYSIWYT, SFL, SENDYS and CONBUG 53

6.0.1 Comparison of the discussed approaches 60

1 Introduction and Scope

Spreadsheets are a popular tool in companies, as their tabular form allows processing of a large quantity of data. Many crucial calculations are computed in these spreadsheets, and it is therefore important that these computations contain no errors. However, research indicates that most large spreadsheets contain non-trivial errors, leading to incorrect results and possible loss of revenues [26].

Substantial research has been conducted to eliminate the problem, either by providing tools to create spreadsheets or by detecting, locating and possibly fixing such errors. The focus of this work is to summarize and compare several existing approaches for the localization of faults in spreadsheets.

A fault localization technique is a system that aids the user in finding the cause of a problem, signified by the difference between an expected behavior and the computed behavior [15]. This means that the user of such techniques must already be aware of the existence of a fault. According to Ruthruff *et al.* [30, p. 216], the goals of fault localization are:

(1) to reduce the search space of the spreadsheet, allowing the user to focus on a few highly suspicious cells and

(2) to rank fault candidates so that cells with a higher fault likelihood are ranked higher than others, therefore "prioritizing the sequence of the users search".

A survey of research on quality assurance in spreadsheets has recently been published by Jannach *et al.* [20], who split the topic into six categories, including a brief overview of fault localization approaches. To define the scope of this work in more detail, we consider some of these categories and how they relate to fault localization.

1. **Visualization-based approaches**: These techniques provide visual feedback to the user, helping to group or distinguish cells and improve readability. While fault localization is also concerned with the presentation of its results, the focus of this paper is the computation of the feedback.
2. **Static code analysis and reports**: Static analysis does not depend on the evaluation of the spreadsheet, simply analyzing the formulas to detect potential faults. This means that faults can be found without additional user input or the observation of faulty behavior. We briefly describe one such system, a type checker called UCheck [4], in Section 4.2.1.
3. **Testing approaches:** These systems facilitate the testing of spreadsheets by providing test coverage, management and generation tools. They are often an important prerequisite for fault localization.

Two additional categories, Model-driven development and Design and Maintenance support, were proposed, but they are concerned with avoiding faults rather than locating them. The goal of this paper is to describe and compare automated fault localization techniques in detail, including information on input, output and the evaluation of the techniques.

Structure There are two basic approaches to fault localization in spreadsheets: trace- and model-based debugging. In Section 2, we discuss trace-based approaches, which analyze the formulas and use the relationship between cells, created by cell-reference, to reason over fault likelihood. This section compares fault localization with WYSIWYT [30], spectrum-based fault localization, short SFL [15] and an approach that searches for the most influential cell, which we refer to as MOSTINFLUENTIAL [7]. Two model-based approaches, EXQUISITE [18] and CONBUG [15], are discussed in Section 3. Model-based debugging uses a system description and observations to find explanations for the fault, where the description is usually done via constraint programming.

There also exist combined approaches, of which we discuss two: In Section 4.1 we examine SENDYS [15], an approach that 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 4.2. While the main scope of this work is fault localization, we also briefly summarize two repair approaches that aim to correct the fault instead of merely locating it, in Section 5. Finally, we conclude this paper with a brief summary and comparison table in Section 6.

In the remainder of this section, we present the criteria used to compare the approaches in Section 1.1. In Section 1.2, we introduce and briefly explain our running examples.

1.1 Comparison Criteria

We compare the different approaches with the aid of four main criteria.

1. **User Input:** This criterion offers information on the complexity of the required and optional user input, meaning how tangible the expected spreadsheet behavior is to the user. Ideally, an algorithm requires only vague knowledge to function (e.g. judging values as incorrect without specifying the expected values) but can improve further with more concrete knowledge (e.g. providing expected values).

2. **Algorithm:** We compare the algorithms of the approaches, the ideas they are based on, their runtime and fault complexity and how the results are computed.

3. **Output:** The type of output and how well it supports the two goals of fault localization as well as its support for single and multiple faults is taken into consideration.

4. **Evaluation:** If they are available, we compare the evaluations of the prototypes, regarding the type of evaluation, which metrics were used, their complexity and results.

1.2 Basic Definitions and Running Example

To explain fault localization techniques in a more accessible manner, we provide a simple spreadsheet example. Example 1.2.1 shows a simplified salary calculation which assigns a bonus if the employee worked more than 15 hours. We explain this example in more detail while introducing some common terminology in the following paragraphs.

	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, with the fault highlighted by a red dashed border in `D2`, where the value 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(C2:D2)
4	Rogers	20	=B4*16	=IF(B4>15; C4 /8;0)	=SUM(C2:D2)
5	Total		*=SUM(C2:C3)*	*=SUM(D2:D4)*	*=SUM(E2:E4)*

(b) The formula view, showing the faulty cell `D2`, highlighted by a red dashed border, which 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 1.2.1: A spreadsheet of a bonus calculation with a single fault, with input cells in blue and output cells in purple font, highlighting the fault with in `D2` with a red dashed border.

Input / Output

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 (output). The concept of variables is also different: A cell can be compared to a variable c and the corresponding formula producing its value $v(c)$. We differentiate between formula cells, which reference other cells, and constant cells. We define input and output cells as follows [15]:

Definition 1.2.1. (*Input, Output*) In spreadsheets, input cells are cells that are referenced in other cells, but do not reference any cells. Output cells are formula cells that reference cells but are not referenced by any cells themselves.

In Example 1.2.1, input cells are displayed with blue font color (`B2:B4`) and output cells are in purple font (`C5:E5`). Note that this definition allows formula cells that do

not reference any other cells to also be considered input cells. As input cells are usually assumed to be correct, possible errors, for example by changing constants in a formula, could be overlooked in this case.

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 this example 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 1.2.1b, 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 1.2.1, as some approaches can only debug single faults and to keep to complexity of the example low. However, spreadsheets may contain more than one fault, leading to a multiple fault complexity of the debugging problem. Example 1.2.2 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` highlighted with a red dashed border.

	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(C2:D2)
4	Rogers	20	=B4*16	=IF(B4>15; C5 /8;0)	=SUM(C2:D2)
5	Total		=SUM(C2:C3)	*=SUM(D2:D4)*	*=SUM(E2:E4)*

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

Example 1.2.2: A spreadsheet of a bonus calculation with three faults, with the faulty references in red. Note that `C5` is no longer an output cell, as it is wrongly 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 an output cell, as it is now erroneously referenced by cell `D4`. This example is used when explaining multi-fault concepts and model-based approaches, as they can be best explained using multiple faults.

Dependency Graphs

To facilitate the understandability of these examples, we will provide partial program dependency graphs for both Example 1.2.1 and Example 1.2.2. These graphs indicate with grey arrows which cells refer to each other and highlight the fault using a red color. Each node represents a formula cell and contains the cell's coordinates and the cell references used in the formula. Please note that we omitted the input cells and references to them to keep the graph as compact as possible.

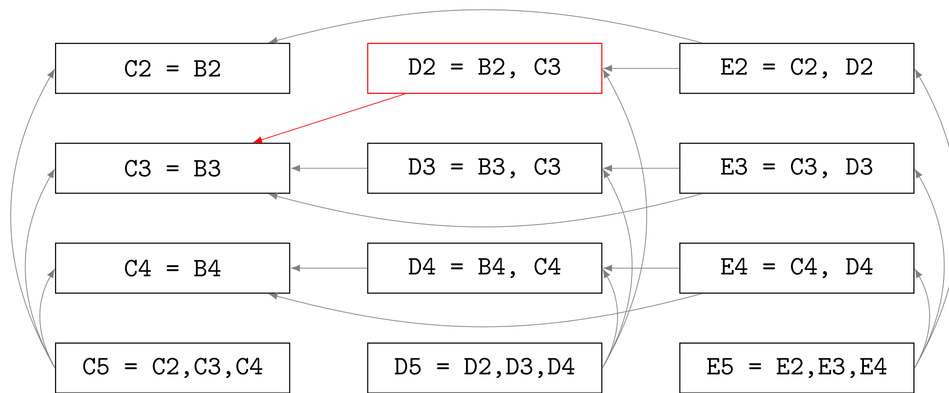


Figure 1.2.3: This partial program dependency graph for Example 1.2.1 shows the cells and their references and the data dependency arrows. The faulty cell `D2` has a red border and the faulty reference to `C3` is indicated by a red arrow.

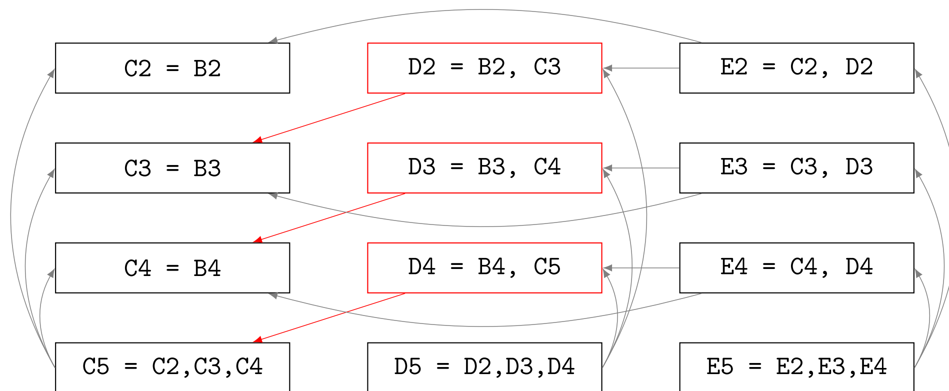


Figure 1.2.4: This partial program dependency graph for Example 1.2.2 shows the data dependency arrows. The faulty references from column `D` to `C` are indicated by red arrows.

2 Trace-based Approaches

Trace-based approaches use data structures inherent to spreadsheets to localize faults, using data dependencies created from cell references in formulas. This section provides an overview of trace-based approaches, including three different techniques for the WYSIWYT system [30], called Blocking, Nearest Consumer and Test Count technique. Additionally, we discuss the approach duped SFL [15] for spectrum-based fault localization as well as MOSTINFLUENTIAL [7], which returns the most influential cell as the one with the highest fault likelihood.

We compare these approaches regarding the criteria proposed in Section 1.1. Section 2.1 examines the input required by the approaches. The underlying ideas behind the approaches as well as the computation of the fault localization feedback are topic of Section 2.2. The output is discussed separately in Section 2.3. Section 2.4 compares the ability of the approaches to handle multiple faults and Section 2.5 discusses the evaluations of the techniques. Finally, Section 2.6 contrasts the approaches broadly with the aid of tables, summarizing the previous sections.

2.1 Input

All fault localization techniques rely heavily on the quality and correctness of the user input. This section describes the type of information needed for the fault localization technique to work. Section 2.1.1 shows different ways expected behavior may be indicated by using so called testing decisions. Section 2.1.2 examines how these testing decisions are processed and combined with the information extracted from the spreadsheet, forming the information base of the algorithm.

2.1.1 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. 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. These testing decisions may take different forms, such as providing expected values, relative assertions or intervals. 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 will refer to as TD^+. Both TD^- and TD^+ are sets of testing decisions, so that $TD = \{TD^- \cup TD^+\}$.

For Example 1.2.1, we provide three testing decisions which can be seen in Figure 2.1.1. 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 (✗).

	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 2.1.1: The testing decisions for Example 1.2.1 with the two positive decisions shaded in green and the negative decision shaded in red.

We now briefly describe the testing decisions and the underlying testing systems used by trace-based approaches.

WYSIWYT

As WYSIWYT was originally developed as a testing methodology, a graphical interface already exists, supporting the user in the creation of testing decisions [29]. This interface allows the user to mark cells with either a check mark or an X mark, in a similar way to our example in Figure 2.1.1. Note that these decisions always judge the values computed by the systems, rather than their formulas. Allowing the user to mark formulas as correct would provide valuable additional information, but raises testing complexity for the user.

SFL

While no graphical user interface was presented for SFL, the creation of test cases works similarly, by marking cell values as expected or unexpected.

Most Influential

The MostInfluential approach is based on a testing methodology called interval testing [7]. The user defines intervals for both input and formula cells, describing the minimal and maximal values a cell may reach. These intervals are propagated through the spreadsheet and a separate bounding spreadsheet is created containing these intervals. The computed values are then checked against their expected intervals and if the limits are violated, the cell is said to have a “symptom of fault”. This is roughly equivalent to the user setting an X mark on the cell, defining its value as unexpected. If a user-defined interval is not violated, the cell value can be described as expected.

The advantage of this technique is that due to the propagated intervals in the bounding sheet, many testing decisions can be inferred from relatively little input. The success is, however, strongly dependent on the position of the provided input: If we provide

intervals for output cells only, these values cannot be propagated further. In contrast, if intervals are provided for all input cells, the values can be propagated to all formula cells in the spreadsheet, allowing for multiple possible input values. Provided that input cells are assumed to be correct, violated intervals should be provided for formula cells with the smallest amount of producers, as they are the nearest to the input cells and usually influence the most cells.

For Example 1.2.1, if we provide an interval of [870 : 880] for the output cell `E5`, no values can be propagated and only `E5` is marked as erroneous. If we provide an expected interval of [30 : 40] for our faulty cell `D2` that is subsequently violated, this symptom propagates down to the output cell `E5`.

Interval testing is similar to assertion testing, where the user may provide additional information about a cell's behavior, for example by setting minimal and maximal values, or comparing its value to another cell in the spreadsheet. Assertion testing can be used in WYSIWYT [8] and the model-based EXQUISITE, which both also support test suites. It is unclear whether multiple input values can also be defined through intervals in these systems.

2.1.2 Algorithm Input

All trace-based approaches use the data flow structures gained from analyzing a spreadsheet's formulas and cell references. MOSTINFLUENTIAL traverses the data dependency graph to reach the most influential cell. WYSIWYT and SFL use slicing techniques [31], which are also based on data dependencies, to create the necessary input for their approaches.

Slicing

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. In our example, `D2` references `B2` and `C3` directly, and `B3` indirectly through `C3`. We refer to this set as *producers* of c. The *consumers*, also called forward slice, of c are all variables that are directly or indirectly dependent on c, in other words use c.

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.

WYSIWYT uses *dynamic* backward slices for its implementation, which contain only cells that actually contribute to the evaluated result of a cell. This means that cell references that do not contribute to the result due to branching conditions are not included in the slice. For the cell `D3` from Example 1.2.1, the static producers would be `B3` and `C3`, but as the condition is not met and `C3` is not needed to compute the value of zero, only `B3` would remain in the dynamic producers.

Hofer *et al.* [15] argue that 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, they use the function $\textsc{Cone}(c)$, which originates from hardware debugging, to compute the set of producers including the cell c. In contrast to WYSIWYT, CONEs are static and contain all cell references, regardless of their true participation in the result.

Test Cases

Fault localization techniques use slicing to combine testing decisions with the data dependency structures, creating test cases. Hofer *et al.* [15] define 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 output cells with their expected values. For trace-based approaches, 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 allows any formula cells with testing decisions to act as outputs for a test case, even if they themselves are referenced.

Additionally, we split a single test case into multiple test cases by creating a new test case $t = (I, \tilde{o})$ for each output cell $\tilde{o} \in O$. This means that every test case t has a single output cell $\tilde{o}$ with a respective 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 can 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 slice 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 is in the slice of $\tilde{o}$. This means that for the given testing decisions for Example 1.2.1, 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, `E5`✗ should be read as the failing test case $t = (I, \texttt{E5})$, where $d(\texttt{E5}) \in TD^-$.

The contribution of cells to each test case can be seen in Table 2.1.1. Note that the input cells in column `B` have been excluded from the test cases, as they are assumed to be correct.

Table 2.1.1: Test case participation marking each cell's participation in the test cases for Example 1.2.1. The test cases are referred to with the output cell and the type of testing decision attached to it, i.e. `E5`✗ 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
E5✗	•	•	•		•	•	•		•	•	•	•
E3✓		•				•				•		
C5✓	•	•	•	•								

Additionally, Figure 2.1.2 visualizes the fact that a cell may contribute to passing as well as failing test cases: Cell C5 is shaded in light green as it participates only in one passing and in no failing test cases. Cells that contribute only to the failing test case $\texttt{E5}_{\text{✗}}$ are shaded light red and cells that contribute to both passing and failing test cases are hatched with both green and red.

	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 2.1.2: Visualization of the test case participation, with cells that participate only in failing test cases shaded red, cells that participate exclusively in passing test cases in green (C5) and participation in both failing and passing test cases marked by green and red hatching.

In procedural programming, inputs are usually passed as function parameters, therefore multiple test cases can be defined easily by passing different values. In spreadsheets, input cells are defined in the same way as formula cells, by entering their value into the cell. This makes it difficult to create multiple test cases for the same variable group, especially for end-users with no knowledge of testing methodology. However, some tools such as WYSIWYT and EXQUISITE support multiple inputs for the same cell and we refer to such testing systems as test suites.

We now have two sources of information: The testing decisions given by the user, judging the values of specific cells, and the slices or 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 that 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. This may happen if the fault is not actually used to compute the value (dynamic vs. static), a computation nullifies the faulty value, for example by multiplying by zero, or an additional fault reverses the effect of the first. This also 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, which are called *oracle mistakes*. This may also lead to an erroneous or faulty cell being marked as correct or a correct cell to be marked as erroneous.

2.2 Algorithms

This section examines how the different approaches handle the mapping of the input or information base to create the fault localization feedback. The form of the output is discussed separately in Section 2.3.

2.2.1 WYSIWYT

Ruthruff *et al.* [30] expanded their WYSIWYT testing methodology by introducing three techniques: the previously developed Blocking technique [27] as well as Test Count and Nearest Consumer. We will describe these approaches in the following paragraphs, starting with Test Count as it is the least complex of the three.

Test Count technique

Test Count uses slicing to reason over test cases and is based on the TARANTULA technique [21]. TARANTULA marks cells that are involved in more failing test cases than passing test cases as more suspicious than others. For cells that participate in at least one failing test case, the fault likelihood is calculated with the following function:

$$\mathrm{FL}_{\mathrm{TC}}(c) = \max(1,\ 2 \cdot |\,\mathrm{TC}_{\mathrm{F}}(c)| - |\,\mathrm{TC}_{\mathrm{P}}(c)|),$$

where $\mathrm{TC}_{\mathrm{F}}(c)$ is the set of failing test cases c contributes to and $\mathrm{TC}_{\mathrm{P}}(c)$ is the set of passing tests c contributes to. Note that cells that do not participate in any failing test case have a zero fault likelihood, whereas all cells that use the formula have a likelihood of at least one due to the maximum function.

Table 2.2.1 shows this fault likelihood calculation applied for Example 1.2.1, returning the highest fault likelihood for the faulty cell `D2` as well as `E2`,`E4`,`E5` and `D4`, as they contribute to one failing test case and no passing test cases.

Table 2.2.1: Fault likelihood values for Test Count, where D2 has the highest possible fault likelihood with four other cells.

c	$\|\mathrm{TC_P}(c)\|$	$\|\mathrm{TC_F}(c)\|$	$\mathrm{FL_{TC}}(c)$
C2	1	1	1
C3	2	1	1
C4	1	1	1
C5	1	0	0
D2	0	1	2
D3	1	1	1
D4	0	1	2
E2	0	1	2
E3	1	1	1
E4	0	1	2
E5	0	1	2

Note that these values are not set in relation to the total number of test cases but rather mapped to likelihood levels, which we are going to discuss in Section 2.3. Due to the small number of test cases given for our example, these results show little distinction and effectiveness.

As participation in passing test cases is given little weight, this approach is robust against coincidental correctness and oracle faults where the user mistakenly places a check mark on an erroneous cell. Errors where the user places X marks in correct cells would in contrast have a noticeable effect on the results and could only be mitigated by placing additional negative testing decisions.

Blocking technique

The Blocking technique uses a similar approach to Test Count, again using test cases created by slicing. In addition to counting the tests, Blocking uses a modified form of the debugging method dicing [10], 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. Cells that participate in both failing and passing test cases are removed from the result. One disadvantage of this method is that positive testing decisions are given a significant weight and are assumed to be correct. This means that oracle mistakes and coincidental correctness are not taken in account, possibly leading to an empty dice or a dice that does not contain the faulty cell. Please note that coincidental correctness may also occur when multiple faults cancel each other out, meaning that this approach may not work for spreadsheets with more than one fault.

The Blocking technique tries to mitigate these issues by using a more conservative approach. Reichwein *et al.* [27] 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. They therefore suggest that cells that contribute to passing and failing test cases should not be eliminated from the search space completely but rather assigned a very low fault likelihood.

This is achieved by distinguishing blocked and unblocked test case participation for each cell. Let us assume a cell c participates in two test cases t_1 and t_2, where the output $\tilde{o}_1$ of t_1 is a producer of $\tilde{o}_2$ of t_2 . The test case t_2 is reachable or unblocked if the testing decisions for $\tilde{o}_1$ and $\tilde{o}_2$ are equal, i.e. both are positive or negative. If the two testing decisions are conflicting, t_2 is blocked for cell c and all other cells that participate in t_1. This means that passing test cases can be blocked by X marks and failing test cases are blocked by check marks.

For example, `E5` from Example 1.2.1 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`$_{✓}$.

The fault likelihood of the Blocking technique is then calculated with the same formula as Test Count, but uses only unblocked test cases for the computation:

$$\mathrm{FL}_{\mathrm{BL}}(c) = \max(1\ ,\ 2 \cdot |\,\mathrm{TC}_{\mathrm{F,U}}(c)| - |\,\mathrm{TC}_{\mathrm{P,U}}(c)|),$$

where $\mathrm{TC}_{\mathrm{F,U}}(c)$ is the set of failing test cases to which c contributes that are unblocked and $\mathrm{TC}_{\mathrm{P,U}}(c)$ has a similar definition for passing test cases. The maximum function once again ensures that any cell that participates in at least one unblocked failing test case has a fault likelihood value of at least one.

As we mentioned above, any cell that contributes to a failing test case should not be removed from the result set completely. If a cell participates only in blocked failed test cases (i.e. $|\,\mathrm{TC}_{\mathrm{F,U}}(c)| = 0$ but $|\,\mathrm{TC}_{\mathrm{F}}(c)| > 0$), its fault likelihood is set to “Very Low”. If a cell does not participate in any failing test cases, blocked or unblocked, its fault likelihood is set to zero or “None”, as in the Test Count technique.

Table 2.2.2 shows the number of passing and failing unblocked test cases and the computed fault likelihood for each cell in Example 1.2.1.

Table 2.2.2: Fault likelihood values for Blocking, with blocked test cases indicated by an appended (B). Discrete values for $\mathrm{FL}_{\mathrm{BL}}(c)$ indicate that the formula was applied, otherwise the likelihood levels were set based on the rules described above. The faulty cell `D2` has the highest likelihood along with four other cells.

c	$\lvert \mathrm{TC}_{\mathrm{P,U}}(c) \rvert$	$\lvert \mathrm{TC}_{\mathrm{F,U}}(c) \rvert$	$\mathrm{FL}_{\mathrm{BL}}(c)$
C2	1	1	1
C3	2	0 (B)	Very Low
C4	1	1	1
C5	1	0	None
D2	0	1	2
D3	1	0 (B)	Very Low
D4	0	1	2
E2	0	1	2
E3	1	0 (B)	Very Low
E4	0	1	2
E5	0	1	2

Note that while the result regarding the faulty cell `D2` is equal to Test Count, the cells with a "Very Low" offer additional distinction and therefore improve prioritization.

At this point, test cases can only be blocked by opposing testing decisions. One possible improvement of this technique would be to allow the effects of a second X mark in the same path to be blocked. Say we have a cell c and a cell p that is a producer of c, and both cells have negative testing decisions (i.e. X marks). The observation that c behaves erroneously is most likely due to the same fault that already influenced the value of p. Such propagated faults could be considered in the calculation.

One advantage of the Blocking Technique is that it needs less test cases than Test Count to create a clear distinction between cells with very low and higher likelihood. While the issues of dicing are lessened by using the "Very Low" likelihood, Blocking is prone to over-emphasize check marks and therefore less robust against coincidental correctness and user errors than the Test Count technique.

Nearest Consumer technique

This approach is the least expensive of the three WYSIWYT techniques, both in runtime and required memory. It uses information from direct consumers instead of slices, which means that only cells that directly reference c in their formula are relevant to its fault likelihood computation. It is unclear whether dynamic references, i.e. only references that are actually followed in the execution, are considered or if all references in the cell are used. As the other WYSIWYT techniques use dynamic slicing, we assume the references are followed dynamically here as well.

The fault likelihood is computed using the following formula:

$$\mathrm{FL}_{\mathrm{NC}}(c) = \mathrm{avg}_{\mathrm{FL}}(DC(c)) + z,$$

where $\mathrm{avg}_{\mathrm{FL}}(DC(c))$ is the average fault likelihood of c's direct consumers. The average is adjusted through z, which is set to one if the number of X marks is larger than the number of check marks in DC, unless there are zero check marks and only one X mark, which is not seen as sufficient evidence.

The average fault likelihood for any set of cells C is defined as:

$$\mathrm{avg}_{\mathrm{FL}}(C) = \frac{\sum_{i \in C} \mathrm{FL}_{\mathrm{NC}}(i)}{|C|},$$

where $\mathrm{FL}_{\mathrm{NC}}$ is the previously computed fault likelihood level or an initial value of zero for cells with positive or no testing decisions. Cells containing X marks have an initial fault likelihood of three, the medium fault likelihood, which is the minimum value for cells with X marks regardless of their average. If a cell has a check mark but the average of its direct consumers is greater than zero, meaning it participates in failing test cases, its fault likelihood level is set to one, emulating the Blocking technique.

Table 2.2.3 shows the iterative computation of the fault localization values: first their initial values, then the setting of the testing decisions and the effect they have on their producers. The adjusted average is then rounded to likelihood levels and used as initial values for the next iteration in the algorithm. $\mathrm{FL}_4(c)$ in the last column shows the final fault likelihood levels for Nearest Consumer with all three testing decisions considered.

The positive testing decision in `E3` emulates the blocking effect by setting the level to one if a check mark is in the cell. Note that the cells in column `E` have a higher fault likelihood than our faulty cell `D2`, as they have `E5` as their direct consumer. This is due to the strong impact the initial fault likelihood has if there exist output cells without testing decisions. In this case, the sum in `D5` is assumed to be correct, but is actually erroneous. Additionally, positive testing decisions for output cells do not have any effect as all are assumed to be correct. This can be observed for column `C5✗`, where the testing decision for `C5` does not change the result any further. Therefore, , either a higher static fault likelihood or the result of other fault localization techniques might serve as better initial likelihoods.

Table 2.2.3: The Nearest Consumer computation, showing the computed averages for each test case and rounding up to likelihood levels in each step $\mathrm{FL}_i(c)$. Columns E5✗, E3✓ and C5✓ should be read from the bottom to the top, as they follow the backward slice of the given cell. The faulty cell D2 receives the second highest fault likelihood along with three other cells.

c	$DC(c)$	$\mathrm{FL}_0(c)$	E5✗	$\mathrm{FL}_1(c)$	E3✓	$\mathrm{FL}_2(c)$	C5✓	$\mathrm{FL}_4(c)$
C2	{E2,C5}	0	1.5	2		2	1.5	2
C3	{D2,E3,C5}	0	1.5	2	1	1	0.8	1
C4	{D4,E4,C5}	0	1.5	2		2	1.7	2
C5	{}	0		0		0	0	0
D2	{E2,D5}	0	1.5	2		2		2
D3	{E3,D5}	0	1.5	2	0.5	1		1
D4	{E4,D5}	0	1.5	2		2		2
D5	{}	0		0		0		0
E2	{E5}	0	3	3		3		3
E3	{E5}	0	3	3	1	1		1
E4	{E5}	0	3	3		3		3
E5	{}	3	3	3		3		3

Note that due to threshold rules such as the one defined above for z, Nearest Consumer also depends on a larger number of failing test cases to function effectively. As for coincidental correctness and oracle mistakes, Nearest Consumer behaves similarly to the Blocking technique and therefore shows the same weaknesses, giving too much weight to positive testing decisions.

2.2.2 Spectrum-based Fault Localization

Similarly to Test Count, SFL uses the data from the test case participation of each cell to compute a similarity coefficient which indicates the 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 2.2.1 shows the hit-spectra matrix for Example 1.2.1 as well as the error vector e.

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. Hofer *et al.* [15] use the Ochiai coefficient for their implementation, which is defined as

$$
\begin{array}{c} \\ \texttt{E5}_{✗} \\ \texttt{C5}_{✓} \\ \texttt{E3}_{✓} \end{array}
\begin{array}{c}
\begin{array}{ccccccccccccc} \texttt{D2} & \texttt{B3} & \texttt{E5} & \texttt{D3} & \texttt{B2} & \texttt{D4} & \texttt{B4} & \texttt{C4} & \texttt{C3} & \texttt{C2} & \texttt{E2} & \texttt{E3} & \texttt{E4} \end{array} \\
\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{pmatrix}
\end{array}
\begin{array}{c} e \\ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \end{array}
$$

Figure 2.2.1: 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.

follows:

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

where $\mathrm{TC_F}$ denotes the set of all failed test cases. 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 a low number of test cases allow high result values, which is not possible for WYSIWYT approaches.

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. The same is not true for WYSIWYT approaches: As the number of test cases is higher and $|\,\mathrm{TC_F}(c)|$ is multiplied by two, the first case would receive a low, the second a very high likelihood.

Table 2.2.4 shows the computed values for each cell in Example 1.2.1, where the total number of failed test cases $|\,\mathrm{TC_F}\,|$ equals one.

Table 2.2.4: Fault likelihood values for SFL using the Ochiai similarity coefficient with $|\,\mathrm{TC_F}\,| = 1$. The faulty cell has a likelihood of one, along with four other cells.

c	$\lvert\,\mathrm{TC_P}(c)\rvert$	$\lvert\,\mathrm{TC_F}(c)\rvert$	$\mathrm{FL_{Ochiai}}(c)$
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

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. Due to the small number of test cases, the WYSIWYT techniques do not offer such confidence in their results, with their highest ranked results ranging in the middle of the fault level spectrum.

2.2.3 Most Influential Cell

Ayalew *et al.* [6] propose fault localization based on the idea of hubs: a cell that is the producer of many erroneous cells has a high likelihood of being faulty. This idea is similar to the Test Count technique and TARANTULA, assuming that a cell that contributes to many failing test cases must be more likely to contain the fault than a cell that only contributes to a few.

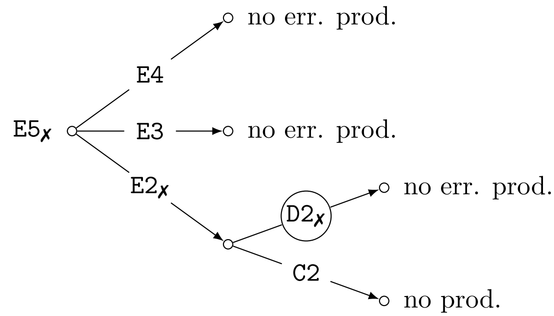
Unlike the other trace-based approaches, MOSTINFLUENTIAL asks the user to select a cell for which to find the most influential producer. The algorithm then traverses the backward slice and calculates a priority status for each cell in the direct producers of the given cell. The priority status is dependent on the number of erroneous producers, as a cell with many erroneous producers has a high likelihood of containing the most influential cell in its backward slice. The cell with the highest priority status is chosen. If two cells have an equal amount of erroneous producers, the number of direct, erroneous consumers is also considered. If these numbers are equal, the algorithm chooses one arbitrarily. Since the bounding sheet, which contains the propagated intervals, provides a potentially high number of testing decisions only direct producers and consumers are considered rather than test cases. The algorithm stops when a cell does not have any erroneous direct producers, assuming the fault is found.

For Example 1.2.1, if we provide intervals for the same three cells that were judged as correct or erroneous, this approach would not yield any result other than the erroneous cell `E5` as the most influential cell. As the intervals are propagated forwards, all consumers of a cell with a testing decision also receive a testing decision. As `E5` has no consumers, the given interval is not propagated. Since the algorithm depends on negative testing decisions to trace the slice of a cell and none are given in the direct producers of `E5`, the algorithm would return `E5`.

However, if we provide an interval for our faulty cell `D2` between 30 and 40, the result of the interval testing would show that `D2`, `E2` and `E5` are below the user expectations and therefore erroneous. Figure 2.2.2 shows how the most influential cell is computed by using a table to show the direct producers and a tree to show the computation steps. Note that erroneous cells are marked with ✗, as only erroneous cells are considered candidates for a solution. If the user requested the most influential cell for `E5`, the algorithm would trace the backward slice up to the faulty cell `D2`.

c	$z \in DP(c)$	$DP(z)$
E5✗:	E2✗	{C2,D2✗}
	E3	{C3,D3}
	E4	{C4,D4}
E2✗:	C2	{}
	D2✗	{C3}

(a) Table showing the producers



(b) Tree showing the computation steps

Figure 2.2.2: The computation for MostInfluential, with erroneous cells marked with an ✗. Figure 2.2.2a shows direct producers for c in $DP(c)$ and their respective producers. The producer with the largest number of erroneous producers is followed, which can be seen in Figure 2.2.2b. The algorithm stops when there are no more erroneous producers left, leaving in this case the faulty cell `D2`.

In comparison to the other approaches, MostInfluential does not process positive testing decisions and returns a single fault candidate instead of a cell ranking. As opposed to using the slice of a cell for the calculation of the priority status, only direct producers and consumers are considered. This necessitates a large number of testing decisions, which can be achieved by providing intervals as close to input cells as possible so that propagation can be used effectively. As we saw for our example, providing intervals for output cells has little effect, as the interval could not be propagated any further. Additionally, please note that the faulty cell might not always be erroneous, either through lack of suitable user input, oracle mistakes or even coincidental correctness. In

this case, the faulty cell would never be detected by this approach.

In fact, a faulty cell f can only ever be returned as the most influential if the user provided a violated interval for f specifically. If testing decisions are made for the producers of f, the intervals would not be violated, as we assume the producers are correct in a single fault situation. In contrast, if violated intervals are provided for the consumers of f, the faulty cell itself would not be marked as erroneous, as the intervals are only propagated forwards.

2.3 Output

Apart from MostInfluential, all algorithms return a numerical value for all cells which indicates their fault likelihood and can be 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. A perfect fault localization technique would rank the faulty cell at the first position. Two or more statements are tied when they have the same value and therefore cannot prioritize the sequence of the search any further. A tie is therefore the set of all cells that have the same fault likelihood. If the true fault is contained in such a set, we speak of a critical tie.

WYSIWYT

The Wysiwyt approaches map the results of their algorithms fault likelihood levels from "None" to "Very High". This mapping is needed as the computed values for Test Count and Blocking are not normalized and a large number of test cases leads to a very high computed value without necessarily improving the confidence in the result. Table 2.3.1 shows how the levels correspond to discrete values, so that the level number and the computed fault likelihood values can be compared.

Table 2.3.1: Fault likelihood values and the corresponding levels the values are mapped to. Note that level one is not used by Test Count and does not correspond to any values as it is set based on rules.

Values	Level	Level Name
0	0	None
	1	Very Low
$1-2$	2	Low
$3-4$	3	Medium
$5-9$	4	High
$10-\infty$	5	Very High

For the Blocking and Test Count technique, any cells that calculate their likelihood through the given equations return at least a "Low" fault likelihood. Blocking and Nearest Consumer additionally use "Very Low" to describe failing test cases blocked by

positive testing decisions, whereas Test Count does not take blocking into account. Note that Ruthruff *et al.* [30] map the Test Count values to levels between zero and four rather than zero and five instead of omitting level one. To simplify the comparison we use the levels described in Table 2.3.1 for all Wysiwyt approaches.

Note that when fault likelihood levels greater than ten are reached, which may happen with as little as five failing test cases, the mappings to the levels do not distinguish any further, therefore losing precision very quickly. Simultaneously, Wysiwyt shows only tentative results when provided with insufficient test cases, as is the case for our running example, which only worsen when the levels are applied.

The mapping and its impact on the visual representation of the result can be seen in Figure 2.3.1, where the computed values for Example 1.2.1 are visualized by shading from light yellow to dark orange for increasing fault likelihood. The graphics to the left visualize the computed values exactly, whereas the graphics on the right show the mapped levels. The faulty cell is highlighted with a dark red dashed border.

	A	B	C	D	E
1		**Hours**	**Salary**	**Bonus**	**Sum**
2	Jones	17	1	2	2
3	Smith	13	0.1	0.1	0.1
4	Rogers	20	1	2	2
5	**Total**		0	0	2

(a) Blocking values

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

(b) Blocking levels

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

(c) Test Count values

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

(d) Test Count levels

	A	B	C	D	E
1		**Hours**	**Salary**	**Bonus**	**Sum**
2	Jones	17	1.5	1.5	3
3	Smith	13	0.75	0.5	1
4	Rogers	20	1.5	1.5	3
5	**Total**		0	0	3

(e) Nearest Consumer values

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

(f) Nearest Consumer levels

Figure 2.3.1: Output visualization for Wysiwyt with cell shading in yellow to orange, where darker colors indicate higher fault likelihood. The value 0.1 is used to indicate very low fault likelihood for the Blocking technique values in Figure 2.3.1a. As Nearest Consumer uses levels from the start, the values in Figure 2.3.1e represent the exact computed levels without rounding.

The comparison of the visualizations shows the loss of information and the creation of larger ties. The size of the critical tie is increased for both Blocking and Test Count, where the size of the critical tie increases from six to ten cells. While Nearest Consumer already takes levels into account for the computation of the average, the rounding of the result still leads to some loss of information: Figure 2.3.1e shows a suspicious increase

in the average for cell C3 due to being referenced by D2 in addition to D2 and C5, which can no longer be observed in Figure 2.3.1f.

One possibility to improve these techniques would therefore be to normalize the exact computed values in a way that would not create more ties. This could be achieved by dividing through the sum of all values or the maximum value reached in the computations.

SFL

When using the similarity coefficient Ochiai, SFL returns values between zero and one. As the Ochiai coefficient also considers confidence, the computed fault likelihood levels are much higher than WYSIWYT, which can be seen in Figure 4.1.1, where the cells with the highest level have the darkest possible shade.

	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 2.3.2: Output visualization for SFL with Ochiai using the same color scale as in Figure 2.3.1. The faulty cell has the darkest possible shade of orange with a value of one.

While the output for SFL still creates ties for cells with the same numbers of failing and passing test cases, the computed values are not modified through a mapping.

Most Influential

While MOSTINFLUENTIAL returns only one result, the most influential cell for a given cell o, the producers of cell o are highlighted, so that the search space of the user is reduced in that regard. Figure 2.3.3 shows how the output could look, assuming a fault likelihood of one for all producers and five for the most influential cell.

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

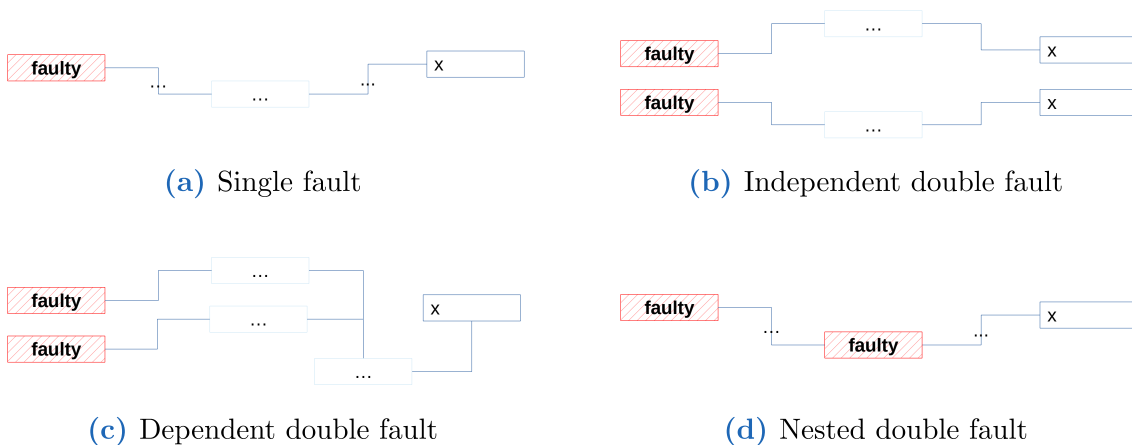
Figure 2.3.3: Output visualization for MOSTINFLUENTIAL using the same levels and color scale as in Figure 2.3.1. The faulty cell D2 is the most influential, as a violated interval was provided for this cell.

While many cells could potentially be tied in this approach, the algorithm simply chooses an arbitrary cell in case of two equally influential cells.

2.4 Fault Complexity

This section compares the number of faults that each approach can handle. Since trace-based techniques are based on heuristics and likelihoods, a single fault complexity is usually assumed and the support for multiple faults is limited. This is also given by the form of the output: MostInfluential returns only a single cell, allowing no feedback for multiple faults and both Wysiwyt and Sfl return 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 2.4.1 provides a schematic visualization of single and double faults and how they are indicated via negative testing decisions. Cells with red hatching are faulty, dots are placeholder for an arbitrary number of intermediate cells, and cells containing an X indicate negative testing decisions.



(a) Single fault (b) Independent double fault (c) Dependent double fault (d) Nested double fault

Figure 2.4.1: 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.

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 one fault. The visual representation of single faults can be seen in Figure 2.4.1a. 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 2.4.1b and Figure 2.4.1c respectively. Figure 2.4.1d shows a subset of dependent faults that are nested. The following sections describe the different types of faults closer and discuss how the approaches handle these faults.

2.4.1 Independent Faults

If two faults do not share any consumers prior to reaching separate negative testing decisions, we speak of independent faults. Given that there exists at least one failing test case for each fault, the faults do not interfere with each other. While currently not supported by any tool to our knowledge, it would be possible to use data-dependency analysis to detect a subset of these faults: If two faults share no common producer, possibly excluding input cells, the existence of these faults could be detected by checking if the intersection of two test cases is empty.

For Example 1.2.2, which has three faults, the faults are independent if separate testing decisions are provided before their paths meet in `E5`. Figure 2.4.2 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 2.4.2: Testing decisions for Example 1.2.2, showing two independent faults. Each test case 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.

As the test cases are created via backward slicing, we can simply follow the dependencies for the cells `E2` and `E4` to obtain the slice. Figure 2.4.3 shows the partial program dependency graph for Example 1.2.1 with the dependencies from `E2` indicated in orange and the dependencies from `E4` in blue.

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. This also demonstrates how any algorithms that use test counting, such as Blocking, Test Count and SFL, would rank cells `C2` and `C3` higher than the other cells, as they participate in the most failing test cases.

WYSIWYT

Multiple faults are supported in principal by WYSIWYT, even if the output form does not highlight multiple faults. As the Blocking technique and Test Count rely on the number of (unblocked) test cases, the fault that has the highest ratio of failing to passing test cases would rank the highest. Since there are only five possible levels, it is also very

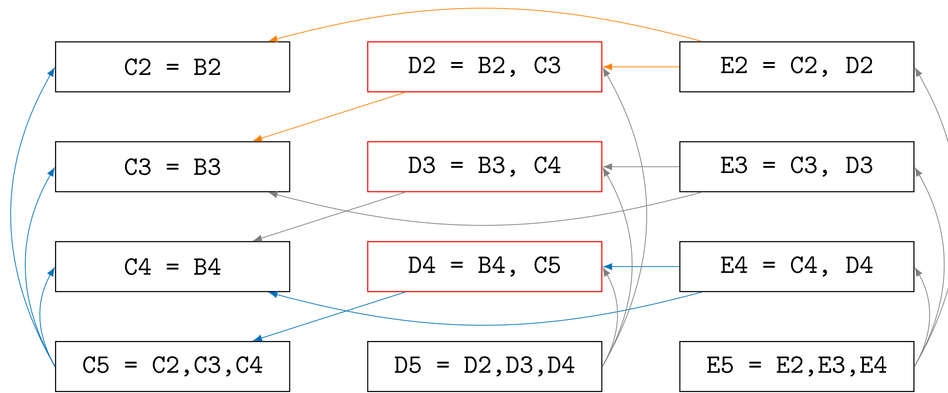


Figure 2.4.3: This partial program dependency graph for Example 1.2.2 shows the data dependencies in dashed arrows, with the dependencies for each test case indicated in blue and orange. The input cells and their references are omitted to keep the graph compact.

likely that both faults receive the same rank. For the two testing decisions defined in Figure 2.4.2, both of these approaches would return a likelihood value of four for C2 and C3, and lower values of two for all other cells in the test case.

The Nearest Consumer technique works with averages and takes testing decisions rather than test cases into consideration. Therefore, , the fault ranking is dependent on the average of the direct consumers as well as X marks placed in the cell itself or its direct consumers. If both of these factors are equal or cancel each other out, it is possible that both faulty cells might have the same rank. For our example, the cells with the testing decisions, i.e. E2 and E4, once again have the highest likelihood level of three. When considering the unrounded values, these levels are immediately followed by 2.25 for C2 with the faulty cells receiving a level of 1.5 each. Note that cell C3 is ranked lower, as it is referenced by E3, for which we provided no test case and is therefore assumed to be correct.

SFL

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 100 % 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 70 %. Therefore, , multiple independent faults would decrease the confidence of the result, allowing other, non-faulty cells to be ranked higher than the faulty cell. 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.

Most Influential

As this approach only outputs a single cell and chooses one arbitrarily if both have the same priority status, it can only ever find one fault at a time. However, if we translate the two negative testing decisions in two separate testing actions the user requests, this approach can point out the most influential cell for each independent fault separately. This means that for our example, when the user requests the most influential cell for `E2` and subsequently `E4`, the faulty cells could be returned as the most influential cell, given that they violate a provided interval.

2.4.2 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 1.2.2, we make the faults in `D2:D4` dependent on this test case. Figure 2.4.4 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 2.4.2, encompassing almost all cells.

	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 2.4.4: Single testing decision for Example 1.2.2, making all three faults dependent on `E5`. All cells that participate in the test case are indicated by red shading.

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.

WYSIWYT

For Blocking and Test Count, dependent faults cause little conflicts. If both faults participate in the same test cases, they have the same fault likelihood level. 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.

Nearest Consumer takes testing decisions and fault likelihood averages into account rather than test cases. In the case of nested faults, the fault f_2, which is closer to the testing decision, is likely to have a higher fault likelihood than f_1 if no additional testing decisions exist. This is due to the initial fault likelihood of all cells being set to "None" and the dilution of the average fault likelihood for cells closer to the input, as they are likely to be used more often. For f_1 to receive a higher fault likelihood than f_2, consumers of only f_1 must have additional negative testing decisions, leading to f_1 contributing to more failing test cases than f_2.

SFL

SFL handles multiple, dependent test cases in the same fashion as the Blocking and Test Count technique. Whichever fault contributes to more failing test cases, 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.

Most Influential

As MostInfluential is not equipped to return a multiple fault result, the fault that influences more erroneous cells is returned. For nested faults, this would be f_1 if test cases were taken into consideration, as it influences all cells that f_2 influences and at least one more: f_2 itself. However, as testing decisions in the form of violated intervals are used to decide the priority status, the fault that has the most erroneous cells in its direct consumers is returned.

2.4.3 Propagated Faults

Another point to consider for the multiple fault complexity is the propagation of faults. As soon as a fault is recognized via a negative testing decision in cell c, it can be assumed that the consumers of c also show erroneous behavior. In the interest of reducing the search space and the complexity of the debugging problem, the consumers of c can therefore be excluded from the relevant cells.

However, as we do not restrict the testing decisions to output cells only, it is possible that a testing decision d_2 may be a consumer of another testing decision d_1. Assuming both testing decisions are negative, the assumed fault complexity becomes relevant: If we assume a single fault, the testing decision d_2 is redundant and only points to a propagated fault, whereas the true fault must lie in the producers of d_1. If we assume multiple fault complexity, it is possible that there lies another fault between d_1 and d_2, influencing d_2 in addition to the fault recognized in d_1. In this case, both testing decisions should be considered in the result.

As the number of faults contained in a spreadsheet is usually unknown to the user and the debugging technique, it is sensible to assume the simplest fault complexity possible.

It can be assumed that the process of debugging a spreadsheet is incremental and after locating and fixing one fault, the user may find other erroneous values and start the process anew.

2.5 Evaluations

Of the discussed approaches, only Ruthruff *et al.* [30] and Hofer *et al.* [15] conducted an evaluation of their prototypes, whereas no evaluation exists for MostInfluential. In the following subsections, we describe the setup of their evaluation, which metrics they used to measure their success and finally which results the evaluations yielded.

2.5.1 Evaluation Setup

WYSIWYT

All three Wysiwyt techniques were evaluated with a combination of user studies and automated testing. Two spreadsheets were used for the experiment, containing five seeded faults that were based on real faults users made in a previous study. Twenty participants were given the faulty spreadsheets with specifications and expected outputs. The study participants were asked to place testing decisions on cells, which were recorded and could then be applied to all algorithms equally.

This setup leads to multiple, different testing decisions for the same spreadsheet and allows insight into oracle mistakes and different testing approaches users might take. Since they also recorded the order of the testing decisions, the effectiveness could be measured at multiple points in the debugging process. This includes measurements for the first and second X mark being placed and the end of the testing sessions, signified by the user changing a formula cell, presumably to correct the fault.

SFL

Sfl was evaluated with an offline batch testing approach, using spreadsheets from the EUSES [12] corpus. For each of the around 1 400 base spreadsheets, Hofer *et al.* [15] created up to five first-order mutants by using mutant 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 study. Finally, 622 single-fault mutants with up to 4 000 formula cells could be used for the evaluation.

The testing decisions for this evaluation were created by comparing the mutated version of a spreadsheet with the original spreadsheet. All output cells that differed from the original were marked as erroneous, while all output cells that were identical to the original were marked as correct. The number of erroneous output cells was surprisingly low, with an average of 1.7 and a maximum of 22. The number of correct output cells

that were provided is significantly larger, with a median of 24 and a maximum of almost 3,000 cells, which could not be provided by a user realistically. It would be interesting to see which of these passing test cases actually influence the result by intersecting with the slice of the erroneous cells.

SFL was compared to several other approaches, including CONBUG and SENDYS, whose results are discussed in Section 3.4 and 4.1 respectively. Additionally, two primitive fault localization techniques were implemented and evaluated: the union and the intersection of the failed test cases $\mathrm{TC_F}$. The union can be compared to slicing, representing all cells that influence one or more erroneous cells, whereas the intersection assumes single-fault properties and returns only the cells that influence all erroneous cells.

2.5.2 Metric

WYSIWYT

Ruthruff *et al.* [30] used two metrics based on the equation

$$\mathrm{eff_{FL}} = \mathrm{avg_{FL}}(C_F) - \mathrm{avg_{FL}}(C_{NF})$$

where C_F is the set of all cells in C that are faulty and C_{NF} is the set of all cells in C that are not faulty. The maximum average for both terms is five, the highest fault likelihood level, and the minimal average is zero, as no negative values are possible. The effectiveness therefore increases with the average number of false positives found.

Table 2.5.1 shows this metric applied to the results for Example 1.2.1 for all approaches that return a ranking. For the WYSIWYT approaches, we normalized the fault likelihood levels to a range of $[0, 1]$ to make them comparable to the SFL results.

Table 2.5.1: Effectiveness metric for trace-based approaches with ranking using normalized likelihood levels for the WYSIWYT approaches, where SFL shows the highest effectiveness for Example 1.2.1.

Approach	$\mathrm{avg_{FL}}(C_F)$	$\mathrm{avg_{FL}}(C_{NF})$	$\mathrm{eff_{FL}}$
Blocking	0.40	0.27	0.13
Test Count	0.25	0.21	0.04
Nearest Consumer	0.40	0.33	0.07
SFL	1	0.67	0.33

As a spreadsheet may contain many cells with a fault likelihood of "None", the $\mathrm{avg_{FL}}(C_{NF})$ might be relatively small. To give more weight to the false positives, two different sets of cells C were considered: For eff_{all}, C comprises all formula cells of a given spreadsheet, while the second metric, eff_{gz}, uses only cells that have a fault likelihood greater than zero for C. This means that eff_{all} shows a higher or equal efficiency than eff_{gz}, as a smaller term is subtracted. If this were not the case, it would be an indicator

for false negatives, which cannot occur as the usage of slices guarantees that the faulty cell is always contained in the result.

The previous tables used the eff_{all} metric, whereas Table 2.5.2 shows the values for eff_{gz}. Note that the effectiveness is significantly lower, even zero for Test Count and Nearest Consumer, showing even less visual distinction between the results.

Table 2.5.2: Effectiveness metric for values greater than zero, showing zero effectiveness for Test Count and Nearest Consumer. Test Count fails to distinguish the faulty cell from the others as all receive the same likelihood and Nearest Consumer has non-faulty cells with a higher rank, leading to an equally high average for non-faulty cells.

Approach	$\text{avg}_{\text{FL}}(C_F)$	$\text{avg}_{\text{FL}}(C_{NF})$	eff_{gz}
Blocking	0.40	0.33	0.07
Test Count	0.25	0.25	0.00
Nearest Consumer	0.40	0.40	0.00
SFL	1	0.82	0.18

One disadvantage of this metric is that no information on the position of the fault(s) in the ranking exist. Non-faulty cells that have lower fault likelihood than the faulty cell are given equal weight in the average as non-faulty cells that rank higher than or equal to the faulty cell. However, any cells that are ranked lower than the faulty cell have little impact on the perceived success of the method, assuming that the user detects and fixes the faulty cell. To reflect this behavior in the metric, we could redefine the definition of false positive to mean only correct cells that have an equal or higher fault likelihood than the faulty cell. For multiple faults, we could either consider all cells before the highest ranked faulty cell (optimistic) or the cells before the lowest ranked faulty cell (pessimistic).

SFL

As Hofer *et al.* [15] only seeded single faults in their mutants, the position of the faulty cell in the ranking can be used to measure the success. Two factors are considered:

1. The absolute rank of the faulty cell, i.e. the number of cells that need to be inspected before reaching the faulty cell.
2. The relative rank of the faulty cell normalized over the number of formula cells, representing the reduction of the search space.

The absolute rank provides the number of false positives the user needs to inspect before reaching the faulty cell. As cells can only receive a likelihood of zero if they do not contribute to any failing test case, all faults that contribute to at least one failing test case are in the result. Similarly to the WYSIWYT metric, a measurement of false negatives is not needed, as the usage of CONEs guarantees that the faulty cell has a likelihood greater than zero.

This metric handles critical ties by assuming the worst case scenario: If one or more cells have the same coefficient value as the faulty cell it is assumed that the user needs to inspect all other cells first. As the primitive approaches do not return a ranking, but rather a set of cells, all cells are assumed to have the same fault likelihood, therefore creating a critical tie.

Table 2.5.3 applies this metric to the results from Example 1.2.1 for the trace-based approaches. It shows the position of the faulty cell in the ranking as well as the size of the critical tie, that is the number of cells with an equal fault likelihood. The absolute rank, assuming the worst case, is the size of the critical tie plus any cells with a higher fault likelihood. The relative rank divides the absolute rank by the number of formula cells in the spreadsheet, in this case twelve.

This metric shows that SFL provides the largest reduction of the search space for our example, leaving only 42 % of cells to be examined by the user. Please note that we used the fault likelihood levels of the WYSIWYT approaches for this metric, which perform worse for this example than their value counterparts. In this case, both Blocking and Test Count would produce the same success rate as SFL when using their computed values instead of levels.

Table 2.5.3: Ranking metric for trace-based approaches, showing both absolute and relative ranks. SFL shows the highest reduction of the search space with the lowest rank. While Blocking had a much higher effectiveness in Table 2.5.1, it has an equal rank to Nearest Consumer in this metric, as higher ranked cells are not taken into account.

Approach	Fault Rank	Critical Tie	Absolute Rank	Relative Rank
Blocking	1	7	7	58 %
Test Count	1	10	10	83 %
Nearest Consumer	4	4	7	58 %
SFL	1	5	5	42 %

As with the WYSIWYT approaches, there may be many formula cells with zero fault likelihood. This metric does not distinguish between zero and non-zero rated cells, instead using all formula cells to compute the relative rank. As it is unlikely that a user inspects all cells in a spreadsheet to find the fault, normalizing the relative ranking over all non-zero cells or the union of CONEs could improve the meaningfulness of this metric.

This metric also looses some accuracy due to critical ties, as the original position of the faulty cell cannot be read from this metric. The position is important because it is better to have a fault with the highest possible likelihood in a critical tie with n cells than to have n cells with a higher rank than the faulty cell. For example, in the worst case, Nearest Consumer and Blocking both have an absolute rank of seven, yet in the best case, the fault could be found immediately for the Blocking result, whereas the user would inspect the three higher ranked false positives first for the Nearest Consumer result.

The information on position and critical ties could be retained by using best, worst and average case scenarios, as has been done for the evaluation of similarity coefficients for SFL [14]. An additional measure of success for rankings would be histograms that show the number of times the faulty cell has the highest possible fault likelihood or using medians rather than averages to diminish the effect of negative outliers.

2.5.3 Results

WYSIWYT

The results of the evaluation conducted by Ruthruff *et al.* [30] showed that the combination of information base and mapping may indeed influence the effectiveness of the techniques. Overall, the Nearest Consumer technique performed best with its mapping consistently showing best effectiveness regardless of the used information base. Note that for few test cases, the Nearest Consumer concept shows higher effectiveness by generally providing higher levels of fault likelihood than its counterparts. As the rank of the faulty cell is not considered in their metric, this increase in effectiveness might not be significant.

Additionally, the robustness of the algorithms to oracle mistakes was tested, but did show significant differences for the single approaches.

SFL

The empirical evaluation by Hofer *et al.* [15] showed that SFL performed better than primitive approaches with no significant increase in runtime. While the primitive approaches intersection and union performed with an average absolute rank of 30.8 and 41.1 respectively, SFL improved on this result with an average absolute rank of 26.3, meaning that 26 cells need to be examined before reaching the faulty cell. SFL produced a relative rank of 20.3 %, leaving only 20 % of all cells to be inspected by the user, as opposed to 22 % and 27.3 % for intersection and union respectively. As the rank-based metric described in the previous subsection was used, these numbers refer to the worst case scenario, where all cells in the critical tie need to be examined before the faulty cell.

2.6 Comparison

To conclude the section of trace-based approaches, we provide a short summary and an overview of the discussed criteria. The Nearest Consumer mapping appears to be the strongest of the three WYSIWYT approaches [30], with a good robustness to oracle faults and high effectiveness. SFL has some similarities to the Test Count technique, but uses a more sophisticated mapping that takes confidence into account. It might be interesting to see these techniques combined with each other by using the SFL result as the initial fault likelihoods, which are currently set to zero for all cells in the WYSIWYT

approaches. MostInfluential only considers negative testing decisions, but does not create a ranking of cells and is therefore the most limited approach of the five.

Table 2.6.1 shows a broad comparison between the Wysiwyt techniques, Sfl and MostInfluential.

Table 2.6.1: Comparison of all trace-based approaches

Approach:	Wysiwyt	Sfl	MostInfluential
User Input			
Testing decisions	values (✓/✗)	values (✓/✗)	expected intervals
Input complexity	low	low	medium
Algorithm			
Fault complexity	multiple	single	single
Runtime complexity	low	low	low
Output			
Type	cell ranking	cell ranking	single cell
Range	$\{0, (1), 2, 3, 4, 5\}$	$[0, 1]$	-

The input complexity indicates how much knowledge the user must have of the spreadsheet, ranging from vague knowledge, which can be indicated by X or check marks, to concrete values. While Wysiwyt systems do not decrease in efficiency with the presence of multiple faults, the presence of multiple faults is not detected or communicated to the user. Comparisons regarding the user and algorithm input are discussed in Section 2.6.1, and Section 2.6.2 shows an overview of the evaluations for the trace-based approaches.

2.6.1 Input

Table 2.6.2 compares the types of user input that is requested for the approaches. The required input complexity indicates how much knowledge the user must communicate to the system in order to return a reasonable result. Usually, a single negative testing decision is sufficient to start up the fault localization process and to reduce the users search space. The result of the fault localization usually improves with the amount of specification provided, which can be done via additional testing decisions, assertions or multiple input values for the same cell.

Table 2.6.2: Comparison of the required and optional user input for trace-based approaches

Approach:	WYSIWYT	SFL	MOSTINFLUENTIAL
Testing decision type	values (✓/✗)	values (✓/✗)	expected intervals
Required input complexity	low	low	medium
Additional assertions	✓	-	-
Multiple input values	via test suite	-	via intervals
Optional input complexity	high	low	medium

Used Information

In Table 2.6.3, we compare how each approach uses the user input to create its information base, i.e. the information needed for the algorithm to execute. Both the Blocking technique and the Test Count technique from WYSIWYT use test cases from slices, SFL uses the test cases created from CONEs. Some approaches use only the testing decisions or use them in addition to the tests. For example, the Blocking technique uses the decisions of the cell values in a test case to determine which cells have a lesser likelihood of contributing to the fault. MOSTINFLUENTIAL and Nearest Consumer do not take test cases into account at all, using the testing decisions of a cell c and its direct producers or consumers instead. All cells whose references are used directly in the formula of a cell c are direct producers of c, while all cells that reference c directly are direct consumers.

To keep the tables compact, we use the following abbreviations: BL for the Blocking technique, TC for Test Count, NC for Nearest Consumer and MI for MOSTINFLUENTIAL. The last row shows that all three WYSIWYT approaches use a dynamic slice or execution trace to determine test case participation, whereas SFL uses static CONEs. It is unclear whether MOSTINFLUENTIAL uses static or dynamic consumers or producers.

Table 2.6.3: Used information bases for all trace-based approaches, indicating the parts of the user input that is used in the algorithm. The last row shows whether dynamic or static references were considered, which is the same for all WYSIWYT approaches and unknown for MOSTINFLUENTIAL (MI).

Approach:	BL	TC	NC	SFL	MI
Positive testing decisions (TD^+)	✓	-	✓	-	-
Negative testing decisions (TD^-)	✓	-	✓	-	✓
Passing test cases (TC_P)	✓	✓	-	✓	-
Failing test cases (TC_F)	✓	✓	-	✓	✓
Direct consumers (DC)	-	-	✓	-	✓
Direct producers (DP)	-	-	-	-	✓
Execution trace	dynamic	dynamic	dynamic	static	-

2.6.2 Evaluation

While Table 2.6.4 shows a brief overview of the evaluation setup, the results cannot be compared in tabular form, as the two evaluations used very different approaches and metrics to measure their success.

Table 2.6.4: Evaluation comparison for WYSIWYT and SFL

Approach	WYSIWYT	SFL
Experiment type	user study (20 participants)	offline (batch testing)
Number of base spreadsheets	2	1,400
Number of faulty versions of spreadsheet	2	622
Number of formula cells	-	up to 4000
Fault origin	user study	mutation operators
Fault complexity	multiple	single
Testing decision location	formula cells	output cells
Input cells assumed correct	✓	✓
Constant formulas are inputs	✓	✓

The advantage of WYSIWYT's evaluation is that they used real-life faults gained from user studies and injected multiple faults in their faulty spreadsheets. The testing decisions were set in the user study, therefore creating a realistic debugging situation. The used metric gives an idea of the visual effect the fault localization has, but does not include the position of the faulty cells or how many cells are ranked higher than the faulty ones.

The Sfl evaluation, in contrast, tested on a large number of spreadsheets with many formula cells, covering a wider variety of spreadsheets. The faulty spreadsheets were created automatically using mutant operators, as were the testing decisions. While the testing decisions were limited to only output cells, it is unclear whether this testing behavior resembles human actions. The metric used gives no indication on the visual distinction between faulty and non-faulty cells or the confidence with which the faulty cell was rated.

Trace-based approaches are based on heuristics and the analysis of the cell references. They offer fault likelihoods for all relevant cells and an intuitively understandable form of output with low runtime. This form of output and in part the algorithms can only handle limited multi-fault problems and often return large ties for cell blocks that are not distinguishable, i.e. participate in the same test cases or use the same testing decisions.

3 Model-based Approaches

In this section we compare two approaches, Exquisite [18] and ConBug [15], which use model-based debugging to locate faults. First, we briefly discuss some basic principles of model-based diagnosis.

Model-based diagnosis or Mbd is a method originally used to debug hardware components, developed by De Kleer and Williams [11] and Reiter [28]. It uses models which consist of the logical description of the system that behaves erroneously and observations about the system to reason over possible faults. The system must consist of components which interact with one another and the goal of Mbd is to isolate the components that are faulty. In the context of spreadsheets, the system is the spreadsheet itself and its components are the cells. The model or system description is created by translating the user-defined expected values, inputs and formulas into constraints. The computed values of cells are observations, contradictions between the observations and the system description mean that a fault exists and blame can be assigned to one or more cells.

As the system descriptions only need to provide positive assertions, i.e. expected values, no knowledge of the system or even the computed values is needed by the user. This is similar to the concept of black box testing.

Two important concepts are needed for model-based debugging:

Definition 3.0.1. (*Conflict Set*) A conflict is the set of components that are involved in the calculation of a violated assumption and therefore cause an inconsistent model.

In other words, they "cannot all be functioning correctly", i.e. at least one of them is faulty [11, p. 102]. To keep the computational complexity as low as possible, only minimal conflict sets are used for Mbd.

Definition 3.0.2. (*Diagnosis*) A diagnosis Δ, also referred to as *candidate* [11], is a set of components that explain the erroneous behavior.

This means that when these components are assumed to be incorrect, the system and the model are consistent, i.e. no assumptions are violated. A predicate $AB(c)$ is used to describe whether a component c is assumed to be *ab*normal. While any supersets of a diagnosis are also diagnoses, only minimal diagnoses are relevant for Mbd to reduce the search space of the user as much as possible.

Conflict sets can be compared to slices, as both create a subset of components, in this case cells, that isolate the faulty behavior and produce the value of the erroneous cell. For a single erroneous variable $\tilde{o}$, any cell in the slice or conflict may be the faulty cell and can be regarded as a diagnosis. A slice created for multiple erroneous cells does not equal a minimal conflict set [32], as one slice might be the subset of another.

The main advantage of MBD as opposed to trace-based approaches is its ability to recognize and communicate multiple faults as well as the sometimes drastic reduction of the search space. However, as the discussed approaches do not yet rank their candidates to reflect their fault likelihood, the search is not prioritized as it is done for most trace-based approaches.

3.1 Input

As model-based approaches need conflict sets as their starting point, a conflict must first exist and be recognized by the user. This is done similarly as in trace-based approaches, via a combination of user input and reasoning over the given information. We shall briefly explain what types of user input the systems allow in Section 3.1.1 and how the conflict sets are created in Section 3.1.2.

3.1.1 User Input

The testing decisions the user provides for MBD approaches differ from trace-based approaches, as MBD accepts a wider variety of inputs than the Boolean check or X mark. We maintain that a negative testing decision $d(c) \in TD^{-}$ is any input given by the user regarding the cell c which render the model inconsistent and therefore points to erroneous behavior. Positive testing decisions TD^{+} add information on expected behavior but are not in conflict with the model.

Required User Input

The input required for both approaches resembles the test case scenario for traditional programming, i.e. $t = (I, O)$, where input values and expected output values are provided by the user. For every $\tilde{o} \in O$ there must exist a testing decision $d(\tilde{o})$. The input is either given by the values in the input cells or it can be entered by the user manually to facilitate test suites with multiple test cases with the same input values.

To start fault localization procedures there must be at least one negative testing decision $d(\tilde{o})$ in the form of an expected value that differs from the computed output. Other testing decisions such as an X mark negating the computed output are not enough information to initiate MBD but may serve as additional input for EXQUISITE.

Optional User Input

The user may also define passing test cases via positive testing decisions. As the expected value is identical to the computed value, these can be set via check marks from the user. EXQUISITE allows setting X marks to indicate a negative testing decision, providing additional constraints to improve the result.

EXQUISITE uses assertions to gain additional information about the model: The user may communicate how two cells should relate to each other or define absolute or relative

values for cells (`MIN`, `MAX`, `EQUAL`). The tool also offers the possibility of creating test suites by providing multiple expected values for the same cell given different input values.

Both MBD approaches assume that the testing decisions provided by the user are correct. The success of the approach is therefore directly dependent on the quality and correctness of the user input.

For our examples, we provide one test case with multiple output values for each example. Regarding the positive testing decisions, we reuse the check marks that `E3` and `C5` received in previous sections, stating their values to be correct.

For Example 1.2.1, which is a single-fault example, we assume the total sum of paid salaries was provided by accounting, stating the expected value of `E5` as \$ 874 instead of \$ 866. This leads to a failing test case $t_1 = (I, O_1)$ with

$$I = \{(\texttt{B2}, 17), (\texttt{B3}, 13), (\texttt{B4}, 20)\} \qquad \text{and}$$
$$O_1 = \{(\texttt{C5}, 800), (\texttt{E3}, 208), (\texttt{E5}, 874)\}$$

providing the expected values for input and output cells. Example 1.2.2 has three faults in `D2:D4`, of which one is masked by coincidental correctness. In this case, we assume that the user has a finer knowledge of the expected output and can provide expected values for `E2:E4`. The failing test case for this example is $t_2 = (I, O_2)$, using the same input values and

$$O_2 = \{(\texttt{C5}, 800), (\texttt{E2}, 306), (\texttt{E3}, 208), (\texttt{E4}, 360)\}.$$

3.1.2 Algorithm Input

While the user input provides the expected behavior of the system, model-based approaches use minimal conflict sets to compute their diagnoses.

Exquisite

The QUICKXPLAIN algorithm [22] is used to compute the conflict sets by using divide and conquer principles: splitting the constraints given by the spreadsheet and the user input and checking their consistency recursively. One of the later improvements in EXQUISITE [19] is the use of dependency analysis to prune the constraint problem. Cells that are in the forward slice of an erroneous cell are not added to the model, as they only propagate the symptom of fault further.

ConBug

The conflicts for CONBUG are created by using the CONEs of the erroneous output cells $\tilde{o}$ where $d(\tilde{o}) \in TD^-$. These cones are identical to the CONEs used for test cases in SFL in Section 2. Conflicts from CONEs are only minimal if we assume that all output cells $\tilde{o}$ are not referenced by any other cell. If we had two cells $\tilde{o}_1, \tilde{o}_2$ where $\tilde{o}_1$ is referenced by $\tilde{o}_2$, there would be two conflict sets where one is the superset of another. Additionally,

it is likely that the erroneous value for $\tilde{o}_2$ is simply propagated from $\tilde{o}_1$, therefore little additional information would be gained from the second test case. One possible solution for this problem would be to prune the conflict sets of the non-minimal conflicts by using a dependency analysis similar to Exquisite.

For Example 1.2.1, the conflict set is given by the one failing test case, comprising the entire range between `C2:E4` and `E5`, i.e.

$$C_{\texttt{E5}} = \{\texttt{E2}, \texttt{E3}, \texttt{E4}, \texttt{E5}, \texttt{D2}, \texttt{D3}, \texttt{D4}, \texttt{C2}, \texttt{C3}, \texttt{C4}\},$$

excluding input cells as they are assumed to be correct. QuickXPlain would also return this conflict set, as these formula cells are needed to compute the erroneous value and can therefore not be split up further.

The failing test case t_2 for Example 1.2.2 provides two conflict sets, which are identical to the test cases for independent faults in Figure 2.4.2. We therefore have

$$C_{\texttt{E2}} = \{\texttt{E2}, \texttt{D2}, \texttt{C2}, \texttt{C3}\}$$

due to the negative testing decision for `E2`, i.e. $d(\texttt{E2}) \in TD^{-}$, and

$$C_{\texttt{E4}} = \{\texttt{E4}, \texttt{D4}, \texttt{C4}, \texttt{C5}, \texttt{C3}, \texttt{C2}\}$$

due to $d(\texttt{E4}) \in TD^{-}$. Note that `C3` and `C2` are in the conflict set because they are producers of `C5`, which is in turn erroneously referenced by `D4`.

The passing test cases have no influence on the conflict sets for ConBug, it is unclear whether they influence QuickXPlain and therefore the conflict sets used for Exquisite. However, the information from positive testing decisions influence the result by adding them as additional constraints to the respective solvers.

3.2 Approach

Both algorithms need to create a model from the given information, combining the user expectations with the computed values and formulas of the spreadsheet.

3.2.1 Exquisite

Exquisite transforms the user input and the formulas of the spreadsheet into a constraint model using Choco[1] as a back-end. As Choco does not fully support the use of Real numbers, this approach is limited to Integer values only.

To compute diagnoses, a modified version of Reiters Hitting Set algorithm [28] is used. Given a set of conflict sets CS, a set H is called a hitting set if there exists a non-empty

[1]Choco: http://www.emn.fr/z-info/choco-solver/

intersection with each conflict C from CS, i.e. $\forall\, C \in CS \,|\, C \cap H \neq \varnothing$. Starting with one conflict set at the root, a new branch is opened for each element in the set. Each node represents a possible diagnosis, so the corresponding formula is removed from the constraint model. If the model is now consistent, an explanation of the fault is found and its node from the HS-DAG tree is closed. If the model cannot be solved, a new conflict set is added to the node, branching out again. The possible diagnosis now consists of two cells, one from the first conflict set, one from the second. Again, the formulas from the cells in the possible diagnosis are removed from the constraint model, checking the model for consistency. It is also necessary to close nodes when a diagnosis was found but a subset already exists as a diagnosis, as we are only interested in minimal diagnoses. Nodes are expanded until they create a valid hitting set or until there are no more conflict sets to add that do not intersect with the current hitting set of the node.

For Example 1.2.1, each element in the conflict set $C_{\texttt{E5}}$ would be seen as a diagnosis and checked for consistency. Figure 3.2.1 shows the hitting set computation for Example 1.2.2 with the conflict sets $C_{\texttt{E2}}$ and $C_{\texttt{E4}}$, with check and X marks indicating whether the set is minimal and the colors of the nodes indicating the consistency with the model.

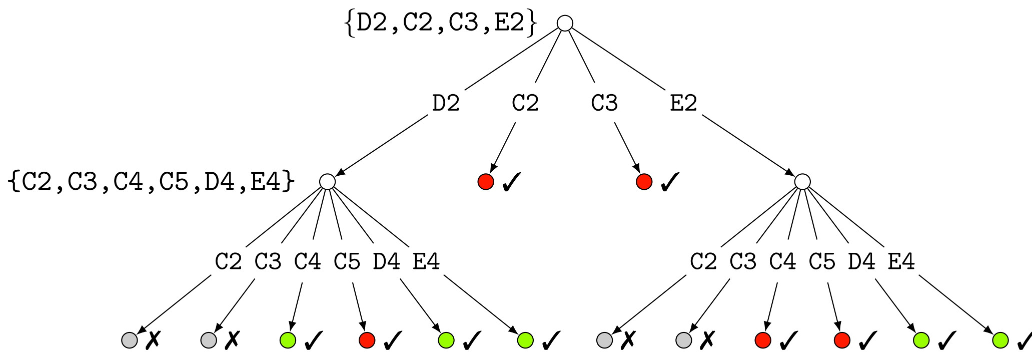


Figure 3.2.1: The HS-DAG to compute the hitting sets for Example 1.2.2, where check and X marks indicate whether the hitting set is minimal or not and the colors of the nodes indicate consistency with the model. Green nodes mean that the hitting set is a diagnosis, red nodes are not consistent and therefore cannot explain the faults and gray nodes do not need to be checked as they are non-minimal hitting sets.

We can see that while {C2} and {C3} are both minimal hitting sets, they do not provide sufficient explanation for the fault and are therefore red. In the second level of the tree, we see that any hitting sets containing C2 and C3 are no longer minimal and do not need to be checked for consistency. Finally, there are four hitting sets of cardinality two that are marked as green, meaning they pass the consistency checks and are therefore minimal diagnoses.

This algorithm was later parallelized, which allowed the tree to be constructed faster, as multiple cores could be used [19, pp. 12-15]. Another improvement was the introduction of trace-based technology via dependency analysis, as already discussed in Section 3.1.2.

3.2.2 ConBug

ConBug uses two different types of solver to compute the diagnoses: constraint solvers Minion[2] and Choco on one hand, and the SMT solver Z3[3] on the other hand. Instead of calculating the diagnoses via hitting sets, ConBug leaves the computation of the diagnoses to the solver back-end. This is achieved by creating a model of the spreadsheet and using auxiliary variables to set the abnormal status of a cell. The problem size can be varied: it is possible to translate all formulas in a spreadsheet, regardless of their relevance, or to reduce the model size and only include cells that contribute to a conflict. As the exact approaches differ for each solver system, we examine the process of the model creation in more detail in the following subsections.

Constraint satisfaction problem

For the constraint model, a variable for each cell is created, linking a cell's formula with its abnormal status. For conditional constructs, several variables are needed. Assuming a formula cell c has the constraint encoding of $f(c)$ for its formula, an auxiliary variable describing its abnormal status can be connected with $AB_c \vee f(c)$. If the cell is deemed abnormal, the constraint of the formula does not have to hold. Additionally, the expected values defined by the user input need to be encoded into the constraint satisfaction problem by also linking them to the cell variables. The constraint model then needs to be solved. The solutions of the auxiliary variables provide the diagnoses, each solution encoding one candidate set.

This snippet of the encoding for Example 1.2.2 shows how the abnormal predicate is linked to the formula of `E2` and the input and expected output is specified for cells `B2` and `E2`:

$$
\begin{aligned}
C_i &:= \texttt{B2} = 17, \\
&\dots \\
C_f &:= AB_{\texttt{E2}} \vee \texttt{E2} = \texttt{C2} + \texttt{D2}, \\
&\dots \\
C_o &:= \texttt{E2} = 306.
\end{aligned}
$$

As neither Minion nor Choco offer complete support for Real numbers, this approach is limited to Integers only.

Satisfiability Modulo Theories (SMT)

Instead of constraint conversion, Außerlechner *et al.* [5] propose to use SMT solvers to debug spreadsheets. Satisfiability modulo theories solvers are based on Boolean satisfiability (SAT) and work with background theories, including support for Real numbers. For this, the algorithm input – including input values, formulas and expected output

[2] Minion: http://minion.sourceforge.net
[3] Z3: http://z3.codeplex.com

values – is encoded in one large formula, connected via conjunctions ($\wedge$), again using auxiliary variables. For SMT solving, the negated form of AB is used and combined with the cell formula, so that $\neg AB_c \wedge f(c)$ holds. For our example, the encoding for E2 and B2 would then be:

$$\texttt{B2} = 17 \wedge \cdots \wedge \neg AB_{\texttt{E2}} \wedge \texttt{E2} = \texttt{C2} + \texttt{D2} \wedge \cdots \wedge \texttt{E2} = 306.$$

To compute the satisfiability of the given model, the solver tries to find possible values for the given variables and predicates. As there exists a conflict, the model cannot be solved, but these values can be used to find diagnoses, which ConBug extracts by using two methods.

One method uses minimal correction sets (MCSes) [24], which return the minimal amount of cells that need to be changed for the model to be satisfiable. For each possible solution, the solver has to be called again with the addition of blocking clauses that contain already found diagnoses which need to be excluded from the search result.

The other method is an improvement over MCSes, MCSes-U [25], which uses the unsatisfiability core to identify a set of components that lead to conflicts. While these conflicts are not necessarily minimal, all variables that are not in the conflict set can be set to $\neg AB = \mathit{true}$, potentially saving computation time.

3.3 Output

Both model-based approaches return a set of diagnoses D, each diagnosis Δ offering a possible explanation of the fault. Diagnoses may consist of a single cell or, in the worst case, contain all cells of a conflict set. This means that multiple faults are well supported and model-based approaches are likely to outperform trace-based approaches in that regard.

Both Mbd approaches return the same output, as they use the same concept of consistency checks for their diagnoses. For Example 1.2.1, five single-fault diagnoses are returned resulting in

$$D_1 = \{\{\texttt{E5}\}, \{\texttt{E4}\}, \{\texttt{E2}\}, \{\texttt{D4}\}, \{\texttt{D2}\}\}.$$

Note that this result is on par with the best outputs produced by the trace-based approaches regarding the position of the faulty cell and shows a further reduction of the search space as there are no cells that are ranked lower than the faulty cell. For Example 1.2.2, Mbd returns four double-fault diagnoses, of which one consists of two of the three faulty cells D2 and D4:

$$D_2 = \{\{\texttt{E2,E4}\}, \{\texttt{E2,D4}\}, \{\texttt{D4}, \texttt{D2}\}, \{\texttt{E4,D2}\}\}.$$

The cell D3 cannot be recognized as faulty by the model-based approaches, as the fault is masked by the if-condition that evaluates to false. While the fault is still present, the cell produces the correct and expected value, which is confirmed by cell E3 producing the expected result. As no conflict is caused by E3, no conflict can be solved by removing E3, meaning this error would go undetected.

Input Cell

The diagnoses returned from EXQUISITE do not contain input cells, as they are assumed to be correct. Jannach *et al.* [20] define input cells as cells that do not contain formulas, which means that constant formulas, i.e. formulas that do not reference other cells, are also seen as components that can behave abnormally. In contrast, Außerlechner *et al.* [5] define input cells to be any cells that do not contain cell references. As CONBUG also assumes input to be working correctly, constant formulas are excluded from the result.

Cardinality

The cardinality of the diagnoses returned from EXQUISITE is limited to the number of conflicts by using the hitting set algorithm. For example, if QUICKXPLAIN only returns one conflict set, the HS-DAG algorithm would allow for single diagnoses only, as there are no additional conflicts to append to a node.

While CONBUG does not share these limitations, if there exist diagnoses of lower cardinality, for example a single fault that could explain the erroneous behavior, only the diagnoses with the smallest cardinality are returned in a first step. For a single test case, a typical single cardinality diagnosis is the erroneous cell itself. This effect can be mitigated by adding passing or failing test cases.

If we provided test case t_1 from Section 3.1.1 instead of t_2 for Example 1.2.2, meaning only one failing test case for a multiple fault example, the single-fault diagnoses from D_1 would be returned for Example 1.2.2. Note that D_1 leaves five cells to be inspected by the user, whereas the union of diagnoses in D_2 shows that only four cells remain, as `C5` can be excluded as a potentially faulty cell.

Form of the Output

The output is a set of diagnoses D, where each diagnosis Δ is a set of cells that explain the fault, therefore D is a set of sets. This means that each diagnosis is equally likely to apply and the search cannot be prioritized. As a large number of diagnoses is possible for MBD, the presentation of the result becomes increasingly difficult with the size of D as well as the cardinality of the diagnoses, $|\Delta|$. Single fault diagnoses could be visualized by highlighting the possibly faulty cells in the spreadsheet, offering reduction of the search space without prioritizing the search. When dealing with multiple fault diagnoses, it becomes necessary to provide a list of the diagnoses, creating the illusion of prioritizing at the risk of confusing the user. If the diagnoses differ in cardinality, it is also possible to rank lower cardinality diagnoses first, as is done in EXQUISITE.

Techniques from WYSIWYT or SFL that return a fault likelihood ranking help to prioritize the search and might therefore perform better for single faults. A combination of the two approaches which offers a ranking of diagnoses are discussed in Section 4.1.

3.4 Evaluation

Both EXQUISITE and CONBUG were evaluated on several occasions, regarding their runtime and performance. In the following subsections, we describe the setup of their evaluations, which metrics were used to measure their success and finally which results the evaluations yielded.

3.4.1 Evaluation Setup

Exquisite

For the first evaluation of their tool [18], Jannach and Engler created multiple versions of a spreadsheet with an increasing number of rows. The spreadsheet modeled a product table and the largest spreadsheet had eight product rows. They then injected "common" faults into the spreadsheet, which are not described in more detail and can therefore not be compared to the mutant operators. In their example, they showed a Reference Replacement error (RFR) as well as Contiguous Range Shrinking (CRS).

They tested against single, double and triple faults, as constraint-based approaches have the potential to excel in multiple fault diagnoses. Interestingly, they only used a single test case, not making use of the ability to combine several test cases, suites or additional assertions.

A later evaluation [17] reused the artificial product table to compare scaleability, but additionally injected single faults in some real life spreadsheets, for example with 224 input cells and 119 formula cells. The domains for the variables were set between 0 and 100,000.

ConBug

An evaluation of CONBUG and a comparison to SFL, SENDYS and primitive algorithms was conducted by Hofer *et al.* [15] and already discussed partly in Section 2.5. Due to Choco's restrictions regarding Real numbers, the CONBUG system could only be used for a subset of the spreadsheets. Only 227 of the 622 mutants were used for this part of the evaluation, containing an average of 219.8 formula cells and up to 2564 formulas in a single spreadsheet.

CONBUG was evaluated again by Außerlechner *et al.* [5], comparing the different solver back-ends regarding their runtime. As MINION and Choco are limited in their ability to handle Real numbers, the evaluation was split into several parts, each part using a different spreadsheet corpus.

To measure runtimes, the constraint solvers MINION and Choco were compared to the SMT solver Z3. An Integer corpus of 33 spreadsheets was used to create 220 mutants, containing single, double or triple faults. On average, the spreadsheets contained 39 formulas, with up to 233 formula cells in one spreadsheet. The constraint solvers MINION and Choco were given a time limit of 20 minutes whereas Z3, which was expected to perform faster, had a limit of 5 minutes. The solving time was averaged over 100 solver calls, and the variable domains were limited to an interval of [-2000,5000] for all solvers.

To measure the effectiveness of ConBug, a different corpus comprising 183 Real numbered spreadsheets based on the EUSES corpus was used, testing the efficiency of the SMT solver on real-world spreadsheets.

3.4.2 Metric

Exquisite

As the focus of their evaluation is the runtime of the tool, Jannach and Schmitz [19] do not use any metrics to measure the success of their approach, simply stating the number of diagnoses returned. For single-fault diagnoses, this number can be compared to the absolute ranking position metric from Section 2.5.2, as all diagnoses are ranked equally. For multiple fault complexities, the cardinality of the diagnoses becomes relevant, as diagnoses with lower cardinality are ranked higher than others. Additionally, this number is not set in relation to the number of cells or formula cells in the spreadsheet. To make Mbd diagnoses comparable to trace-based results, the union of the diagnoses could be considered as the number of cells remaining to be inspected.

ConBug

Hofer *et al.* [15] use the same ranking metric already described in Section 2.5 to measure how ConBug performs against trace-based algorithms. Außerlechner *et al.* [5] measure the success of their approach with the formula

$$\text{eff}_{\text{mbd}} = 1 - \frac{|cells\ in\ diagnoses|}{|cells\ in\ spreadsheet|} \cdot 100.$$

This metric states the reduction of the problem size, meaning what percentage of cells the user can safely exclude from the search of the fault. This corresponds inversely to the relative ranking position metric from Section 2.5.2.

3.4.3 Results

Exquisite

For the spreadsheet with various sizes of a product table, the original tool (without improvements) returned exactly one diagnosis, the correct one. Runtime for the first prototype was quite slow, with eight rows of products already taking 22 second for a single fault diagnosis. The runtime could be improved later due to "better problem encoding, branching heuristics, more efficient data structures and other engineering efforts" [17].

The parallelized version with dependency analysis resulted in an improvement of 70 % in runtime against the original implementation of their tool (run with newer hardware and library versions) [19, pp. 16-24]. Multiple faults could be recognized and do not necessarily lead to a larger number of diagnoses. However, the algorithm does sometimes return a large number of diagnoses, which are not ranked by the current implementation.

This evaluation also showed the influence the position of the fault has, as faults injected in output variables took longer to find than faults injected in an arbitrary formula cell. This is presumably due to the size of the producers, as an output cell possibly depends on more cells to calculate its value than other formula cells. The same can be observed for complex formula cells, as they might have a larger number of cell references and therefore a larger number of producers. Additional data, such as the number of producers for example, would be needed to confirm these assumptions.

Another effect that could be observed from this evaluation was the influence of domain sizes, with the reduction of domain size from $[0, 100\,000]$ to $[0, 10\,000]$ often leading to longer runtimes. This is due to the solver having a smaller set of values to assign to variables, leading to more difficulty in finding a valid solution.

ConBug

The comparison with trace-based approaches [15] showed that on average, ConBug performed slightly better than the primitive union of Cones approach, which was the least successful of all approaches. The average relative ranking for ConBug was 27.9 %, between 27.5 % for the intersection and 29.3 % for the union. As opposed to the intersection, ConBug provides support for multiple faults, whereas the intersection might return an empty set. Another reason for ConBug's poor performance might be the average number of provided testing decisions. While the average test case in the larger corpus contained two erroneous output variables, the smaller Integer corpus had an average of 1.2 variables.

In this first evaluation, ConBug, running with Choco as a back-end, had a runtime of 361.7 milliseconds, performing ten times worse than Sendys (see Section 4.1), the slowest of the compared approaches. The runtime concern was addressed by Außerlechner *et al.* [5], as the use of an SMT solver showed drastic improvements: On average, Z3 was six times faster than the constraint solvers Minion and Choco. Additionally, the constraint solvers produced a large percentage of timeouts (36 % and 11 % of spreadsheets respectively), as opposed to Z3, which only timed out in less than 5 % of all cases even though its limit was much shorter.

Differences in performance with the problem size were also measured by comparing the time it took to solve a problem with only Cones as input compared to converting the entire spreadsheet. While reducing the size of the problem yielded shorter solving time, MCSes-U was more often forced to time out when using cones instead of the entire spreadsheets.

As SMT solvers have the additional advantage of supporting Real numbers, the second part of the evaluation compares Z3 performance for Integer and Real spreadsheets, showing that Real number solving takes on average 2.6 times longer than Integer solving. Finally, ConBug with SMT solving on real-world spreadsheets including Real numbers showed that the search space for the user could be reduced by 80 % on average, with a median of 97 %.

Unfortunately, the results from the first evaluation cannot be compared directly to the results from the second, as a different corpus was used. However, the reduction of

the search space of 80 % would be equivalent to a relative ranking of 20 %.

3.5 Comparison

We conclude the section about model-based fault localization with a summary and comparisons between Exquisite and ConBug. The two approaches differ mainly in the technologies used as back-ends and the computation of the diagnoses. Table 3.5.1 shows a broad comparison of the two approaches.

Table 3.5.1: Comparison of the model-based approaches

Approach:	Exquisite	ConBug
User Input		
Testing decisions	expected values	expected values
Input complexity	high	high
Algorithm		
Conflict sets	QuickXPlain	Cones of TD^{-}
Diagnoses calculation	hitting set	AB predicate
Solvers	Choco	Choco, Minion, Z3
Fault complexity	multiple	multiple
Runtime complexity	high	high
Restrictions	Integer-only	-
Output		
Type	set of diagnoses	set of diagnoses
Ranking	by cardinality	by cardinality

For model-based approaches, the distinction between positive and negative testing decisions is less important, as test cases are created by providing expected outputs. A failing test case is any test case that provides an expected output that differs from the computed output. Exquisite is more versatile regarding the user input, allowing assertions and X marks to create additional constraints but still requires at least one failing test case to create conflicts. Additionally, Exquisite supports test suites, allowing for multiple test cases with the same input cells. See Table 3.5.2 for a comparison of the two approaches regarding their input.

Table 3.5.2: User input for model-based approaches

Approach:	Exquisite	ConBug
Required input complexity	high	high
Failing test cases (expected output)	✓	✓
Passing test cases (✓)	✓	✓
Optional input complexity	very high	high
Additional assertions (including ✗)	✓	-
Multiple input values	✓	-

3.5.1 Evaluation Comparison

The evaluations for Exquisite almost exclusively measured the runtime of the tool as opposed to the performance. While the fault was always found, the number of diagnoses was sometimes quite large, which violates the fault localization goal to prioritize the search space. The spreadsheets used to evaluate Exquisite are smaller in terms of number of formula cells, which is up to several thousand formulas for ConBug. The restrictions of the domain size were set to $[0, 100\,000]$ for Exquisite, allowing only positive values, and $[-2\,000, 5\,000]$ for ConBug. While the runtimes cannot be compared between the approaches, ConBug's comparison between Choco and Z3 showed significant improvement in runtimes with the additional advantage of lifting the Integer restriction.

4 Combined Approaches

As we have shown in the previous sections, both trace-based and model-based approaches have their weaknesses. Some efforts have been made to combine different methods of fault localization to compensate some of these weaknesses. In this section, we briefly discuss SENDYS [15], a combination of SFL and a light-weight model-based approach, and the research done by Lawrance *et al.* [23], who present a combination of WYSIWYT fault localization with the fault detection system UCheck. In favor of a direct comparison, we examine these approaches separately, as they have little similarities.

4.1 Spectrum ENhanced DYnamic Slicing (Sendys)

SENDYS combines SFL and light-weight model-based debugging [16], using diagnoses created from the hitting set of slices.

4.1.1 Algorithm

User Input

SENDYS uses the same user input as SFL, computing similarity coefficients which are later used as initial fault probabilities. For the model-based part, SENDYS uses the CONEs from the negative testing decisions as conflict sets, just as CONBUG does.

Diagnoses

To produce diagnoses, hitting sets [28] are constructed from the conflict sets CS. SENDYS uses the same definition and construction technique as described in Section 3.2.1 for EXQUISITE, where H is a hitting set if $\forall C \in CS \mid C \cap H \neq \varnothing$. As opposed to EXQUISITE, every minimal hitting set is also a minimal diagnosis for SENDYS, as there is no constraint solver used to check consistency.

This means for Example 1.2.1 that the CONE of `E5` is the conflict set CS, and each of its ten elements is returned as a diagnosis, i.e.

$$D_1 = \{\{\texttt{C2}\}, \{\texttt{C3}\}, \{\texttt{C4}\}, \{\texttt{D2}\}, \{\texttt{D3}\}, \{\texttt{D4}\}, \{\texttt{E2}\}, \{\texttt{E3}\}, \{\texttt{E4}\}, \{\texttt{E5}\}\}.$$

Regarding Example 1.2.2, without the solver back-end, two single-fault diagnoses, {`C2`} and {`C3`}, are returned as they contribute to both conflict sets. Additionally, eight double-fault diagnoses can be found with the hitting set algorithm.

$$\begin{aligned} D_2 = \; & \{\{\texttt{C2}\}, \{\texttt{C3}\}, \{\texttt{D2,C4}\}, \{\texttt{D2,C5}\}, \{\texttt{D2,D4}\}, \\ & \{\texttt{D2,E4}\}, \{\texttt{E2,C4}\}, \{\texttt{E2,C5}\}, \{\texttt{E2,D4}\}, \{\texttt{E2,E4}\}\}. \end{aligned}$$

Likelihood Computation

To be able to rank the diagnoses, the probabilities of the cells in a diagnosis Δ are multiplied with the counter probabilities of all cells that are not in the diagnosis. In this case, these a priori probabilities are created from the normalized coefficients returned by SFL,

$$P(c) = \frac{\text{SFL}_{\text{Ochiai}}(c)}{\sum_{i \in C} \text{SFL}_{\text{Ochiai}}(i)},$$

resulting in the formula

$$\text{FL}_{\text{SENDYS}}(\Delta) = \prod_{c \in \Delta} (P(c)) \cdot \prod_{c' \notin \Delta} (1 - P(c')).$$

The fault likelihoods of the diagnoses are then mapped back to the individual cells to create a ranking. The mapping function can be written as follows:

$$\text{FL}_{\text{SENDYS}}(c) = \sum_{\Delta \in D(c)} \text{FL}_{\text{SENDYS}}(\Delta),$$

where $D(c)$ is the set of diagnoses that contain c. To create probabilities, these values are once again normalized by dividing by their sum.

Table 4.1.1 shows how the fault likelihood values are computed for Example 1.2.1. Due to the single conflict set, the hitting set algorithm is not used at all, which means that the prioritization of the cells is identical to SFL. However, due to the mapping process and the computation of probabilities, the likelihoods for SENDYS are much lower for all cells. Note that the column $\text{FL}_{\text{SENDYS}}(\Delta)$ denotes likelihoods for diagnoses rather than cells. While these are equal to cells in this example, they could not be displayed this easily in the case of multiple cardinality diagnoses.

Table 4.1.1: The computation steps needed to compute the fault likelihood values for Sendys for Example 1.2.1. The column $\text{FL}_{\text{Sendys}}(\Delta)$ shows likelihoods for diagnoses rather than cells, which can only be displayed in the case of single cardinality diagnoses. The probability $P(c)$ is normalized by the coefficient sum and $\text{FL}_{\text{Sendys}}(c)$ is normalized by the sum of the diagnosis likelihoods.

c	$\text{Sfl}_{\text{Ochiai}}(c)$	$P(c)$	$1-P(c)$	$\text{FL}_{\text{Sendys}}(\Delta)$	$\text{FL}_{\text{Sendys}}(c)$
C2	0.7	0.083	0.917	0.032	0.08
C3	0.6	0.071	0.929	0.027	0.07
C4	0.7	0.083	0.917	0.032	0.08
C5	0	0	1	-	0
D2	1	0.119	0.881	0.047	0.12
D3	0.7	0.083	0.917	0.032	0.08
D4	1	0.119	0.881	0.047	0.12
E2	1	0.119	0.881	0.047	0.12
E3	0.7	0.083	0.917	0.032	0.08
E4	1	0.119	0.881	0.047	0.12
E5	1	0.119	0.881	0.047	0.12
Sum:	8.4	1		0.39	1

For Example 1.2.2, Sendys suffers from the same issue as the trace-based approaches, as shared producers from both faulty cells allow C2 and C3 as single diagnosis. In the approaches in Section 3, these diagnoses are filtered out, as they do not sufficiently explain the erroneous cells. As Sendys has no such capabilities, they remain as single diagnoses and therefore receive the highest rankings.

Output

The most interesting contribution of this approach is the process of mapping fault likelihoods to diagnoses and then back to cells: this allows the combination of any model-based approach with any approach that returns likelihoods for cells. While the interval of the likelihoods returned by Sendys is $[0, 1]$, which is equal to the Ochiai coefficient in Sfl, the values will typically be much lower, as the sum of the values must be equal to one, whereas the similarity coefficients do not have such restrictions.

Figure 4.1.1 shows how the output for Sendys could be represented to the user. As these results could hardly be distinguished using a color scale from zero to one, the scale was adjusted to shade the highest fault likelihood in the darkest orange.

Additional information could be gained by skipping the last step of the mapping, leaving ranked diagnoses instead of ranked cells. The advantage of this would be that in case of multiple faults, the correlation between two cells can be made clear, whereas this information is lost in the cell ranking.

	A	B	C	D	E
1		**Hours**	**Salary**	**Bonus**	**Sum**
2	Jones	17	0.08	0.12	0.12
3	Smith	13	0.07	0.08	0.08
4	Rogers	20	0.08	0.12	0.12
5	**Total**		0	0	0.12

Figure 4.1.1: Output visualization for Example 1.2.1 for Sendys with the darkest orange showing the highest fault likelihood. The shading is adjusted to account for the much smaller levels, using 0.12 as the maximum possible likelihood. The faulty cell is ranked the highest along with four other cells.

4.1.2 Comparison

In this section, we compare Sendys to the trace-based approaches from Wysiwyt, Sfl and the model-based approaches, in particular ConBug, see Table 4.1.2 for a rough comparison.

Table 4.1.2: Comparison between Wysiwyt, Sfl, Sendys and ConBug

Approach:	Wysiwyt	Sfl	Sendys	ConBug
User Input				
Testing decisions	value (✓/✗)	value (✓/✗)	value (✓/✗)	expected value
Input complexity	low	low	low	high
Algorithm				
Fault complexity	multiple	single	multiple	multiple
Runtime complexity	low	low	low-moderate	high
Output				
Type	ranking	ranking	ranking, diagnoses	diagnoses
Range	$\{0, (1), 2, 3, 4, 5\}$	$[0, 1]$	$[0, 1]$	-

The evaluation that compared Sendys with Sfl, ConBug and primitive approaches has already been discussed in Section 2.5. In this section, we only examine the results pertaining to Sendys. Sendys showed an increase in runtime over the purely trace-based approaches due to the hitting set calculation, but regarding effectiveness, it had the best results for both absolute and relative ranking.

Sendys improves on Sfl by providing support for multiple faults, albeit with the same limitation as Exquisite: if only one conflict exists, Sendys and Exquisite can only return diagnoses of cardinality one. However, Exquisite checks the possible diagnoses for their consistency and only allows a diagnosis if the model without the diagnosis would not cause a conflict anymore. Sendys relies only on the fault probabilities from Sfl to provide adequate results.

Sendys has been compared to an early version of ConBug, outperforming ConBug narrowly in effectiveness and widely in runtime. While Sendys is slower than Sfl and primitive approaches, its average runtime was ten times faster than that of ConBug,

running with the Choco back-end. Another advantage over ConBug is that Sendys does not depend on expected values for erroneous variables.

4.2 UCheck and WYSIWYT

In 2006, researchers from the Oregon State University developed a system that shared reasoning regarding faults in spreadsheets. The prototype used the type checker UCheck and the WYSIWYT fault localization system to evaluate how a combination of the two techniques would perform and whether the heuristics of the combination played a role in performance [23]. In the following subsections, we briefly describe the UCheck system, how the combined approach works and how it was evaluated.

4.2.1 UCheck

Abraham and Erwig [2] developed a tool called UCheck for type checking which automatically infers information about data types from labels. Labels are often found in spreadsheets to describe the kind of data contained, for example the names of the employees in Column `A` as well as the headers in the first row in Example 1.2.1. These automatically inferred headers can be applied to cells containing numerical data. When users reference these cells in their formulas, the type is checked for consistency. If the types of the values used in a formula are inconsistent, a possible fault is detected and communicated to the user. This is done by shading the affected cell in orange, and cells that would propagate the fault further, the cell's consumers, would be shaded yellow. This behavior makes sense for a fault detection tool, as it is pointing out errors to the user. In fault localization, highlighting propagated faults would cause unnecessary confusion: The cause of the propagated fault lies somewhere else, and while it is possible that the cell itself is also faulty, this fault would have to be recorded and handled separately.

UCheck should be able to detect all three faults in Example 1.2.1, because for example in `D3`, `[HoursWorked|Jones]` and `[TotalSales|Smith]` create a mismatched type. The ability to detect this fault depends on how conditions are handled.

The advantage of this technique is that it requires no additional user input, only requiring that users make use of labels to describe their data. Its success is dependent on the quality of the labels used to describe the data, although incorrect header inferences did not lead to false unit errors in their preliminary evaluation [2, 1]. It is also relevant that UCheck is only able to detect reference errors [23], as changes in constants for example would not result in changed types.

4.2.2 Combining UCheck and WYSIWYT

The combined reasoning system is very similar to the WYSIWYT system, which had been adapted for Excel Spreadsheets instead of Forms\3 in [9].

Input

While no information is given regarding the algorithm input, we can assume that testing decisions (✓/✗) and the resulting test cases were used as an information base. As UCheck

does not require user input outside the traditional spreadsheet environment, it can easily be integrated into the WYSIWYT testing methodology.

Mapping

Each system has its own mappings internally, with WYSIWYT returning values between 0 and 5 and UCheck assigning high fault likelihood to detected type errors and lower likelihood for propagated faults. As we already pointed out in Section 4.2.1, we believe propagated faults are unsuitable fault localization results as they are unlikely to contain the true fault. It is unclear which of the three fault localization techniques described in Section 2.2 performs the mapping for the WYSIWYT part.

The mapping of the shared reasoning system takes place in the reasoning database, which takes the output of the different fault localization techniques and returns either:

- the lowest, (*Combo Min*)
- the highest (*Combo Max*) or
- the average (*Combo Average*)

of the ratings returned from each system. Additionally, these mappings were either used with the raw feedback from each system ($\{0, ..., 5\}$ for WYSIWYT, $\{0, 1, 2\}$ for UCheck) or the least likely results were filtered out with a threshold. As it is unclear how the raw results were scaled, we assume that UCheck's adapted values are treated as $\{0, 2.5, 5\}$. For the threshold mapping, the propagated type errors returned by UCheck are dropped from the result and only immediate type errors are considered. For WYSIWYT, this means that cells with a very low fault likelihood are dropped from the result, while the remaining cells are considered to have a value of 5. We are uncertain of the advantage of such a drastic measure for WYSIWYT, as this leads to loss of information and the creation of larger ties.

4.2.3 Evaluation

Evaluation Setup

For the evaluation, 932 mutants were created by seeding single faults into the same spreadsheet, representing the different mutation operators described in [3]. For the simulation of the testing decisions, a probability model was used that acted as the user. It inserted oracle faults with a higher likelihood of misdiagnosing wrong cells as correct than vice versa. The combined reasoning was compared with the algorithms performing on their own as well as the threshold mapping compared to the raw data mapping.

Metric

The metric used to measure the success of the techniques was the same as for the WYSIWYT system, described in Section 2.5.2, which measures the avg_{FL} of all faulty cells and subtracts the avg_{FL} of all non-faulty cells in the evaluation. As we assume

single faults, C_F only consists of one cell. Therefore, a negative value would indicate a false negative, i.e. the fault was not found.

$$\text{eff}_{\text{FL}} = \text{avg}_{\text{FL}}(C_F) - \text{avg}_{\text{FL}}(C_{NF})$$

As both averages are displayed in the evaluation, the metric also shows the average likelihood of the false positives (correct cells with a fault likelihood greater than zero) that were found. While we know that C is the set of formula cells of a spreadsheet, it is unclear whether only cells with a fault likelihood greater than zero were used, or if all formula cells are contained in the set.

Similarly to our concerns regarding the metric from SFL, we believe that the number of cells in set C should be reduced to include only the relevant cells. As the previous improvement over this metric, which uses only cells with a fault localization value greater than zero, may lose validity with the threshold mapping, we propose to only consider cells that contribute to a negative test case for the set C.

Results

The results showed that the combined reasoning with *Combo Max* performed best with an effectiveness of 1.6, followed by UCheck alone, *Combo Average* and then WYSIWYT alone. While *Combo Min* performed worst, showing the least effectiveness, Lawrance *et al.* [23] propose to use this mapping for a very conservative approach. In the same order, but with a higher effectiveness, the threshold mapping consistently performed better than their raw data counter part for this metric. This is due to the higher values assigned to faulty cells, resulting in higher values for the first term. A slight increase in the second term indicates a low number of false positives, but without further data regarding the number of cells in C_{NF} this information is not sound.

The evaluation also showed that the types of faults were relevant to the success of the approach, especially with UCheck's narrow fault detection capability. As only a single spreadsheet was used and mutated, it stands to reason that the success of UCheck is due to a well maintained and labeled spreadsheet. It would be interesting to see how well the system performs with a wider range of spreadsheets.

While many of the previously discussed evaluations used mutated spreadsheets based on the same mutant operators, no data regarding the correlation between effectiveness and fault types exist. This information would be a valuable addition to the existing evaluations, showing strengths and weaknesses of the respective approaches and allowing comparisons.

5 Repair Approaches

While the focus of this work are fault localization approaches, we briefly describe two repair approaches: GoalDebug [1] and a mutation-based repair approach [13]. While repair approaches produce possible fault fixes rather than the location of the fault, we include this topic as they show many similarities to fault localization. For one, they require the same user input and operate with the same information base as fault localization (i.e. testing decisions, assertions). They also face similar issues regarding the presentation of their results, as they may produce a number of fixes that need to be ranked. It might therefore also be of interest to see how evaluations are executed in this area.

GoalDebug

GoalDebug is a technique that allows end users to debug their spreadsheets by providing assertions on error cells and receiving a ranked list of repair suggestions. The tool internally converts the spreadsheet and the user assertions into constraints and creates a list of possible repairs through change inference. These suggestions are then ranked based on heuristics, allowing solutions that require only a small change to a formula to rank higher than solutions requiring extensive changes.

The approach was evaluated using a handful of spreadsheets with seeded errors [4]. The evaluation focused on the comparison of the original prototype from 2005 and an improved version, but also provided information on the types of faults that could be fixed. The different types of faults or mutation operators were explained in more detail by Abraham and Erwig [3]. The evaluation showed that some mutations could be reversed for all tests, while others where only partially or not at all successful.

Mutation-based Repair

Another repair approach based on genetic programming was proposed by Hofer and Wotawa [13]. This approach uses the same inputs as ConBug, meaning testing decisions that mark cells as correct or incorrect and set expected values for incorrect output values. They create mutants of a faulty spreadsheet and check if the mutated version performs better, i.e. has less failing test cases TC_F than the original spreadsheet. If the number of failed test cases in the mutant is zero, a possible repair mutant has been found. They reduce the search space by allowing changes only to cells that are in the union of Cones from TC_F. Hofer and Wotawa [13] allow mutations on a formula level, making changes to constants, references, function operators or built in spreadsheet functions, for example

`MIN` to `MAX`. As opposed to GoalDebug, the paper does not mention how the possible repairs could be communicated back to the user.

The evaluation of this technique used much of the same setup as the evaluation of Sfl, Sendys and ConBug. 555 spreadsheets from the EUSES corpus were used, and each spreadsheet was tested 16 times. Computation times varied from 2 milliseconds to 40 minutes, but with an average of 16.3 seconds. For 55 % of the spreadsheets, a repair could be found in some of the 16 runs, with 23.6 % of the spreadsheets being able to find a repair approach for every run.

Repair approaches are the next step after fault localization: ideally, the user would not have to find the fault in a formula, but is proposed a list of changes, as is already done for a small set of typing errors in Excel. However, these approaches usually have a high complexity and runtime. It is also necessary to rank the repair suggestions so that the user is not overwhelmed with choice - similarly to prioritizing results in fault localization. GoalDebug provides a possible solution to this problem by ranking its result based on heuristics, the mutation-based repair approach does not discuss such a ranking. Fault localization might be used to improve on fault repair by limiting and prioritizing the cells that might be changed.

6 Conclusion

In this paper, we discussed trace- and model-based approaches as well as combination approaches. One approach combined trace-based technologies with light-weight model-based debugging, and the other combined trace-based approaches (WYSIWYT) with fault detection. Additionally, two repair-based approaches were discussed briefly, allowing for a rough comparison of fault localization techniques. Table 6.0.1 summarizes the differences in these approaches.

Table 6.0.1: Comparison of the discussed approaches

Approach:	Trace-based	Model-based	Combined	Repair
User Input				
Testing decisions	value (✓/✗)	expected values	value (✓/✗)	expected values
Required complexity	low	high	low	high
Optional complexity	high	very high	low	very high
Algorithm				
Based on	heuristics	explanations	heuristics, diagnoses	explanations
Fault complexity	mostly single	multiple	multiple	multiple
Runtime complexity	low	high	low-moderate	high
Output				
Type	ranking	diagnoses	ranking	repairs

6.1 User Input

The first point of comparison is the user input for each approach, its required and optional complexity, amount and quality.

6.1.1 Complexity

Trace-based approaches usually require "less precise" input, often only asking for check and X marks judging a cell's value. By "less precise", we mean the clarity of the mental model the user has of a spreadsheet: Presumably it is easier to mark a value as incorrect rather than providing the expected value. For assertion-based testing decisions as well as interval testing, the user may provide ranges of possible cell values, therefore allowing less precision than an exact value but more precision than simply marking a value as

incorrect. Ideally, an approach should require less precise inputs to function correctly, but support advanced users by allowing higher precision (via assertions) as optional input, increasing the quality of the result. Note that the trace-based approaches discussed in this paper do not have the ability to process these expected values in their fault localization calculation.

In contrast, model-based approaches are well equipped to include any additional assertions, whether they are expected values, ranges or comparisons to other cells. As both model-based approaches as well as repair approaches use solver back-ends, the inclusion of such additional constraints should be relatively simple, although ConBug does not currently offer this functionality.

All discussed approaches use testing decisions on the basis of values, which means that the user judges the correctness of a cell's value, not its formula. While it is usually assumed that the formula producing a correct value is also correct, this need not always be the case due to coincidental correctness. Allowing the user to mark a cell's formula as correct in addition to judging the value could allow further improvements.

6.1.2 Size and Quality

In addition to the complexity of the user input, the number of testing decisions provided is also relevant to the success of the approaches. Intuitively, one would assume that if the user provides more information, the fault localization result would be better, both in size and accuracy. However, preliminary research in this area showed that this was not always the case [14]. Due to coincidental correctness, testing decisions that mark a cell's value as correct may lead to a poorer performance than if the cell was left unmarked.

Additionally, research on oracle mistakes [30] showed that users were less hesitant to mark a cell as correct even if the value was incorrect, than to mark a correct cell as incorrect, which occurred less often.

It can therefore be concluded that while multiple testing decisions are needed for most algorithms to work, more testing decisions do not necessarily lead to a better result and great care must be taken with the setting of the testing decisions.

6.2 Algorithms

The greatest advantage of model-based approaches over trace-based approaches is that the algorithms themselves aim to find explanations for the fault, rather than relying on heuristics and probabilities. While correct cells may sometimes have a very low fault likelihood in trace-based results, they cannot be excluded from the result completely. For model-based approaches, cells that are not contained in any diagnoses do not cause a conflict and cannot be faulty. This means that, potentially, a larger number of cells can be removed from the users' search space for model-based approaches.

6.2.1 Fault Complexity

Trace-based approaches return fault likelihoods and usually assume single faults to simplify the problem complexity. This is also reflected in their output, which can not properly visualize multiple fault complexities. Model-based approaches support multiple faults in their computation and their results by returning diagnoses.

6.2.2 Runtime Complexity

While runtime complexity was not discussed in detail and could not be compared between the different approaches, trace-based approaches generally have the lowest complexity and therefore lowest runtimes. Model-based approaches use more complex algorithms, such as the hitting set algorithm, increasing runtime. The biggest factor in the slow runtime for model-based approaches is, however, the requirement for solver calls, where using SMT solving is currently the fastest possibility. The combined approach Sendys has a significantly lower runtime than the pure model-based approaches due to the omission of solver calls, but performs slower than trace-based approaches due to the hitting set computation.

6.3 Output

The possible output for fault localization is either a single cell, a ranking of cells sorted by their fault likelihood, or a set of diagnoses, usually ranked only by cardinality, where each diagnosis consists of one or more cells that explain the fault. While a single result is the ideal, such certainty is rarely the case for real-life spreadsheets. A ranked list is therefore a good result for single faults as it can be easily be presented visually and prioritizes the search space for the user. Multiple fault diagnoses are more difficult to convey to the user, as they do not have an obvious visual representation. A combination of lists and cell-highlighting might help to reduce the search space for the user.

6.4 Summary

In this paper, we compared trace-based, model-based and combined fault localization approaches and showed their advantages and weaknesses. Trace-based approaches offer more intuitive output while using less complex user input, and might therefore be a good option for novice users. However, these approaches have difficulties handling multiple faults and are less precise than other techniques. In contrast, model-based approaches can handle a wider variety of input but expect more user knowledge in the form of expected values and are therefore better suited for more experienced users. They are well equipped to deal with multiple fault complexities and offer fault explanations instead of likelihoods, but the output is less intuitive and the runtime is much higher than for the trace-based approaches. Sendys combines trace-based debugging with light-

weight model-based diagnoses while being significantly faster than pure model-based approaches, but cannot entirely overcome the problems that multiple faults cause.

With this overview we have identified and compared several strong and weak points of the algorithms. For some weaknesses like the optional user input and the handling of independent as well as propagated faults, we found possible improvements which we will develop further in future research. With the groundwork laid by the combined approaches, it is also possible to combine the model-based approaches with a priori probabilities allowing the ranking of diagnoses. Finally, the knowledge of different evaluation and measurement techniques allows us to improve current metrics to measure the success of the developed approaches.

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. Abraham and M. Erwig, "UCheck: a spreadsheet type checker for end users," *Journal of Visual Languages and Computing*, vol. 18, no. 1, pp. 71–95, Feb. 2007, ISSN: 1045-926X, DOI: 10.1016/j.jvlc.2006.06.001.

[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, M. Clermont, and R. T. Mittermeir, "Detecting errors in spreadsheets," in *Proceedings of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000, pp. 51–63, [Online]. Available: http://arxiv.org/abs/0805.1740 (visited on 04/09/2014).

[7] 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).

[8] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace, "End-user software engineering with assertions in the spreadsheet paradigm," in *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, May 2003, pp. 93–103, ISBN: 0-7695-1877-X, DOI: 10.1109/ICSE.2003.1201191.

[9] J. Carver, I. Fisher Marc, and G. Rothermel, "An empirical evaluation of a testing and debugging methodology for excel," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE)*, 2006, pp. 278–287, ISBN: 1-59593-218-6, DOI: 10.1145/1159733.1159775.

[10] 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.

[11] J. De Kleer and B. C. Williams, “Diagnosing multiple faults,” *Artificial Intelligence*, vol. 32, no. 1, pp. 97–130, Apr. 1987, ISSN: 0004-3702, DOI: 10.1016/0004-3702(87)90063-4.

[12] 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.

[13] B. Hofer and F. Wotawa, “Mutation-based spreadsheet debugging,” in *IEEE 24th International Symposium on Software Reliability Engineering (ISSRE) - Supplemental Proceedings*, 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] D. Jannach, A. Baharloo, and D. Williamson, “Toward an integrated framework for declarative and interactive spreadsheet debugging.,” in *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2013, pp. 117–124, [Online]. Available: http://ls13-www.cs.uni-dortmund.de/homepage/publications/jannach/Conference_ENASE_2013.pdf (visited on 06/29/2014).

[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 and T. Schmitz, “Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach,” *Automated Software Engineering*, pp. 1–40, Feb. 12, 2014, ISSN: 0928-8910, 1573-7535, DOI: 10.1007/s10515-014-0141-7.

[20] 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.

[21] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, 2002, pp. 467–477, ISBN: 1-58113-472-X, DOI: 10.1145/581339.581397.

[22] U. Junker, "QuickXplain: preferred explanations and relaxations for over-constrained problems," in *Proceedings of the 19th National Conference on Artifical Intelligence (AAAI)*, 2004, pp. 167–172, ISBN: 0-262-51183-5, [Online]. Available: http://dl.acm.org/citation.cfm?id=1597148.1597177 (visited on 10/18/2014).

[23] 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.

[24] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *Journal of Automated Reasoning*, vol. 40, no. 1, pp. 1–33, Jan. 2008, ISSN: 0168-7433, DOI: 10.1007/s10817-007-9084-z.

[25] M. H. Liffiton and K. A. Sakallah, "Generalizing core-guided max-SAT," in *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2009, pp. 481–494, ISBN: 978-3-642-02776-5, DOI: 10.1007/978-3-642-02777-2_44.

[26] 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).

[27] 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.

[28] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, Apr. 1987, ISSN: 0004-3702, DOI: 10.1016/0004-3702(87)90062-2.

[29] G. Rothermel, L. Li, C. DuPuis, and M. Burnett, "What you see is what you test: a methodology for testing form-based visual programs," in *Proceedings of the 20th International Conference on Software Engineering (ICSE)*, Apr. 1998, pp. 198–207, ISBN: 0-8186-8368-6, DOI: 10.1109/ICSE.1998.671118.

[30] 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.

[31] 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).

[32] F. Wotawa, "On the relationship between model-based debugging and program slicing," *Artificial Intelligence*, vol. 135, no. 1-2, pp. 125–143, Feb. 2002, ISSN: 0004-3702, DOI: 10.1016/S0004-3702(01)00161-8.