# Fragment-Based Diagnosis of Spreadsheets

Thomas Schmitz[1], Birgit Hofer[2], Dietmar Jannach[1], and Franz Wotawa[2]

[1] TU Dortmund, Germany
{firstname.lastname}@tu-dortmund.de
[2] Graz University of Technology, Austria
{bhofer,wotawa}@ist.tugraz.at

**Abstract.** Large spreadsheets are often difficult to understand and to test. Detecting the true cause of an observed wrong calculation outcome in a chain of calculations is even more challenging. In this work, we propose a novel approach that automatically decomposes large spreadsheets into smaller units called fragments. This decomposition serves two purposes. First, it allows us to apply fault localization procedures that can exploit such structural abstractions to find possible explanations for the wrong outcomes (called diagnoses). This results in a faster identification of the diagnoses. Second, it makes the testing process better manageable for the users, as they can provide simpler test cases to reduce the number of possible explanations of the fault. An empirical evaluation of our method shows that the required running times for computing the possible explanations can be measurably reduced when applying the proposed fragmentation approach and that fragment-based test cases help to significantly reduce the number of possible explanations.

**Keywords:** Fault Localization, Spreadsheet Fragmentation, Model-Based Diagnosis

## 1 Introduction

Spreadsheets are widely used in companies for a variety of purposes, e.g., budget planning, forecasting, price calculations, and investment decisions. One might think that spreadsheets which are used for decision-making are well-tested and free from faults, but the reality is disappointing as newspapers regularly report on financial losses because of faulty spreadsheets. A recent article in The Wall Street Journal [23] informs about a $ 100 million loss of the software company Tibco that was caused by a spreadsheet fault. In addition, the consulting company F1F9 lists twelve famous examples of faulty spreadsheets ("The Dirty Dozen") [6]. Furthermore, Schmitz and Jannach recently published a set of faults found in the spreadsheets of the Enron emails [21].

These examples demonstrate that faults in spreadsheets are a common problem and that many faults remain undetected even when domain experts inspect the spreadsheets. But even when a user detects a wrong calculation outcome in a spreadsheet, the process of visually tracing back the dependencies of calculations to the faulty formula(s) can still be cumbersome for several reasons: (1) crossing

dependency arrows are confusing, (2) the tracing has to be enabled cell-by-cell, which results in significant user effort for large spreadsheets, and (3) dependency tracking between worksheets is not possible [9]. Several automated or semi-automated approaches have been proposed to help the user to find the cause of an unexpected calculation outcome, see [18] for a recent overview.

One of these automated debugging approaches is based on the principles of Model-Based Diagnosis (MBD). MBD is applicable in situations where the user is able to specify the expected values for the output cells. Jannach and Schmitz [17] and Abreu *et al.* [1] have proposed MBD-based approaches for spreadsheet debugging which are capable of automatically identifying sets of formulas which can in principle be responsible for the observed faulty calculation outcomes. The sets of faulty formulas that can "explain" the wrong outcomes are called diagnoses. Unfortunately, "pure" MBD approaches have certain limitations when it comes to huge spreadsheets, as the number of diagnoses and the diagnosis time grows with the number of formula cells in the spreadsheet.

In this paper, we propose to use a hierarchical diagnosis process which allows us to apply MBD techniques to larger spreadsheets. The main rationale of our approach is that we first diagnose the problem at a coarse-grained level. To do so, we automatically partition the faulty spreadsheet into a set of smaller units, so-called fragments [16]. In the first phase of the hierarchical MBD process, these fragments represent the smallest "diagnosable units", i.e., there are fewer possible reasons for an unexpected outcome, namely the fragments, than when every single cell would be considered. The result of the high-level diagnosis process are those fragments that can be the cause of the problem. In the next phase, we present the fragments that explain the faulty outcome, i.e., the diagnoses, to the user. The user can then specify additional test cases for these small fragments to further isolate the true problem. Given these additional test cases, we apply the MBD technique on the more fine-grained level of the individual cells, which finally leads us to the true cause of the problem, i.e., the faulty formulas.

The main contributions of this paper are (i) a fragmentation approach to automatically partition a spreadsheet into smaller structurally connected parts (Sect. 4.1), (ii) an algorithm that applies hierarchical diagnosis techniques on the level of fragments (Sect. 4.2), and (iii) an empirical evaluation demonstrating the advantages of our approach in terms of computational efficiency (Sect. 5).

## 2 Motivating Example

The spreadsheet illustrated in Fig. 1 computes the velocity and the distance covered by an object within three phases (acceleration, constant velocity, and deceleration). The three formulas in row 5 are faulty. The spreadsheet creator has forgotten to divide the computed distance by two, which results in a triple fault comprising the cells B5, C5, and D5. In fact, the developer only made one mistake, but by copying the formulas the number of actually faulty formulas was tripled. At some stage, the user of the spreadsheet realizes that there is a problem, because the accumulated distance remains constant from phase 2 (cell

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | | Acceleration | Constant Velocity | Deceleration | Final state |
| 2 | Initial Velocity | 0 | 6 | 6 | 0 |
| 3 | Acceleration | 2 | 0 | -2 | |
| 4 | Duration | 3 | 4 | 3 | |
| 5 | Distance | 18 | 24 | 0 | |
| 6 | Accum. Distance | 18 | 42 | 42 | |

(a) Value view

| | | | | | |
|---|---|---|---|---|---|
| 2 | Initial Velocity | 0 | =B2+B3*B4 | =C2+C3*C4 | =D2+D3*D4 |
| 3 | Acceleration | 2 | 0 | -2 | |
| 4 | Duration | 3 | 4 | 3 | |
| 5 | Distance | =B2*B4+B3*B4*B4 | =C2*C4+C3*C4*C4 | =D2*D4+D3*D4*D4 | |
| 6 | Accum. Distance | =B5 | =B5+C5 | =B5+C5+D5 | |

Legend: Input cell, Interim cell, Output cell

(b) Formula view

**Fig. 1.** Running Example: Velocity and distance calculation.

C6) to phase 3 (cell D6). When the user manually computes the values for the accumulated distance, he/she realizes that the values for the cells B6 and C6 are incorrect and the values for the cells D6 and E2 are correct.

When the user starts the traditional MBD process with the test data shown in Fig. 1 to locate the cause of this problem, eleven potential causes are computed: {B6, C6}, {B5, D6}, {B5, D5}, {B6, D6, C5}, {B6, D5, C5}, {B5, E2, D2}, {B5, C6, C5}, {B5, E2, C5, C2}, {B6, E2, B5, C2}, {B6, E2, C5, C2}, {B6, E2, C5, D2}. Given only one single test case, it can easily happen in MBD-based approaches that the system returns many combinations of formulas that could explain the faulty behavior. In our example, one of the reported diagnoses ({B5, D5}) is a subset of the true fault. This diagnosis does not comprise the cell C5, as the formulas of B5 and D5 could be changed in a way that would already result in the expected output values for the single test case provided by the user.

Overall, the example shows that basic MBD approaches have limitations in certain situations. Specifically, in our situation two improvements are desirable: (1) the number of diagnoses should be reduced so that the user has to inspect a smaller set of possible causes and (2) the diagnosis {B5, D5} should also contain the cell C5 to indicate that the user should change all of these formulas.

Besides being computationally more efficient, our proposed fragment-based decomposition approach helps us in both mentioned dimensions. Let us assume that the spreadsheet was automatically divided into the three fragments {C2, D2, E2}, {B5, C5, D5}, and {B6, C6, D6}. When using MBD on the fragment level instead of the cell level, only two fragments are reported as diagnoses, namely the fragment {B5, C5, D5} and the fragment {B6, C6, D6}. This in turn means that we can omit the fragment {C2, D2, E2} from further considerations. We can now ask the user to provide a simple test case containing only the cells {B5, C5, D5}. Given such an additional test case, applying the MBD procedure on the fine-grained level will immediately return the true cause {B5, C5, D5} as the only candidate. In this small example spreadsheet, all cells of the fragment are the true cause, but this is not necessarily the case for other spreadsheets.

## 3 Preliminaries

As Model-Based Diagnosis is a formal and logic-based approach, a logical framework for spreadsheets is required. In this section, we first describe the basic concepts of spreadsheets more formally before we explain the MBD process.

The smallest unit of a spreadsheet is a cell. A cell contains either a formula or a value. Formula cells use expressions and references to compute values. We refer the reader to [13] for more information about the structure of these expressions. For this section, it is sufficient to distinguish between formula and value cells.

**Definition 1 (Formula Cells).** *The function formulaCells($C$) returns the set of formula cells contained in a set of cells $C$.*

**Definition 2 (References).** *The function ref($c$) for a formula cell $c$ returns all cells that are directly referenced in the formula of $c$ [13].*

Cells are arranged in a matrix. Therefore, they have a unique position and they can be accessed by their coordinates.

**Definition 3 (Coordinates).** *The function $x(c)$ returns the column index of cell $c$. The function $y(c)$ returns the row index of cell $c$.*

There are two ways of referencing cells, A1 notation and R1C1 notation. In this paper, we use R1C1 notation for identifying cells with equivalent formulas.

**Definition 4 (R1C1 Notation).** *The relative position of a cell $c$ with respect to another cell $c'$ is indicated as '*`R[`$y(c) - y(c')$`]C[`$x(c) - x(c')$`]`*'. A formula expression with relative positions is called a formula in R1C1 notation. Absolute references to a cell $c$ are indicated as '*`R`$y(c)$`C`$x(c)$*' in R1C1 notation.*

**Definition 5 (Copy Equivalence).** *Two cells $c, c'$ are copy-equivalent if they have the same formula in R1C1 notation.*

When two cells are copy-equivalent it does not necessarily mean that their formulas have been copied, but it is a good heuristic to determine that these two cells are semantically equivalent.

*Example 1.* The formula of cell C2 of our running example from Fig. 1 is '`=B2+B3*B4`' in A1 notation and '`=R[0]C[-1]+R[1]C[-1]*R[2]C[-1]`' in R1C1 notation. The cells C2, D2, and E2 are copy-equivalent as well as B5, C5, and D5.

We distinguish between input, interim and output cells:

**Definition 6 (Input, Interim and Output Cells).** *The function input($C$) for a set of formula cells $C$ returns all cells that are referenced by formulas in $C$ but that do not belong to $C$. A cell $c \in C$ is called an interim cell for a set of formula cells $C$ if there exists at least one formula cell in $C$ that references $c$. Otherwise $c$ is called an output cell for $C$.*

$$input(C) = \bigcup_{c \in C} ref(c) \setminus C.$$

$$output(C) = \{c \in C | \nexists c' \in C \text{ where } c \in ref(c')\}.$$

A spreadsheet $S$ consists of a set of formula cells $O = \mathit{formulaCells}(S)$ and a set of input cells $I = \mathit{input}(O)$. Labels and cells which are not referenced by others are not relevant in our approach.

*Example 2.* The spreadsheet in Fig. 1 comprises the following sets of cells: $O = \{\text{C2, D2, E2, B5, C5, D5, B6, C6, D6}\}$ and $I = \{\text{B2, B3, C3, D3, B4, C4, D4}\}$. The output cells are a subset of the formula cells: $\mathit{output}(O) = \{\text{E2, B6, C6, D6}\}$.

If a system under observation does not behave as expected, one can use Reiter's Hitting Set Tree algorithm [20] to determine the possible reasons for the differences between the expected and the observed behavior. We use Reiter's basic definitions and framework to describe the general ideas of MBD.

**Definition 7 (Diagnosable System).** *A diagnosable* system *is a pair* (SD, Comps) *where* SD *is a system description (a set of logical sentences) and* Comps *represents the system's components (a finite set of constants) [20, 17].*

In the context of spreadsheets, the set Comps comprises the spreadsheet's formula cells and SD describes the logic of the formulas. To model whether a formula is assumed to be correct or not, the abnormal behavior is represented in SD with a unary "abnormal" predicate ab(.).

*Example 3.* For our running example, we have $\text{SD} = \{\text{ab(C2)} \vee \text{C2} = \text{B2}+\text{B3}*\text{B4}, \text{ab(D2)} \vee \text{D2} = \text{C2} + \text{C3} * \text{C4}, \ldots, \text{ab(D6)} \vee \text{D6} = \text{B5} + \text{C5} + \text{D5}\}$ and $\text{Comps} = \{\text{C2}, \text{D2}, \text{E2}, \ldots, \text{D6}\}$.

A diagnosis problem arises when a set of logical sentences Obs, which contains input values and expected output values of the spreadsheet, is inconsistent with the computed output of the system (SD, Comps).

*Example 4.* In our example, the set Obs contains the input values shown in Fig. 1(b) and the expected values for the output cells, i.e., $\text{Obs} = \{\text{B2} = 0, \ldots, \text{E2} = 0, \text{B6} = 9, \text{C6} = 33, \text{D6} = 42\}$.

**Definition 8 (Diagnosis).** *Given a diagnosis problem* (SD, Comps, Obs)*, a diagnosis is a minimal set* $\Delta \subseteq$ Comps *such that* $\text{SD} \cup \text{Obs} \cup \{\text{ab}(c) | c \in \Delta\} \cup \{\neg\text{ab}(c) | c \in \text{Comps} \backslash \Delta\}$ *is consistent [20, 17].*

A diagnosis therefore corresponds to a minimal subset of the formula cells which, if assumed to be faulty, explains the system's faulty output, i.e., the system description without these formulas is consistent with the expected values in Obs. To calculate the diagnoses, Reiter proposes the Hitting Set Tree [20] algorithm. In our work, we translate the spreadsheet into a constraint satisfaction problem [17] and use Reiter's algorithm in combination with QuickXplain [19]. Details about our specific algorithm implementation can be found in [17].

## 4 Fragment-Based Diagnosis

In this section, we first describe an evolutionary fragmentation algorithm and then explain how MBD can be adapted to work on the fragment level.

### 4.1 Fragmentation Process

The fragmentation process proposed in this paper is based on the initial ideas presented in previous work [16]. While this previous work only introduced the basic idea, we now describe the fragmentation process itself. The main rationale is that we start with each cell forming its own fragment. These single-cell fragments are easy to understand. However, such a fragmentation does not reduce the complexity when searching for the possible causes of the problem, as too many fragments have to be considered. Therefore, we combine these small fragments to larger ones in an evolutionary process, which consists of two major parts: (1) collapsing copy-equivalent formulas and (2) merging fragments. When we collapse copy-equivalent formulas, all of them are represented by only one formula for which the user has to ensure the correctness. In contrast, merging fragments does not reduce the number of formulas to test but combines connected formulas into a group that can be tested together. While the process of collapsing the copy-equivalent formulas follows strict rules, we use a genetic approach for merging formulas into fragments. The goal of the genetic approach is not to merge as many fragments as possible, but to create fragments with a reasonable size and complexity. We will explain both the collapsing and the merging part below in detail, but first, we formally define fragments.

**Definition 9 (Fragment and Fragmentation).** *A fragment $f$ is a set of formula cells $f \subseteq formulaCells(S)$. The set of formula cells of a spreadsheet $S$ is partitioned into $n$ disjunct fragments $f_i$, i.e., $formulaCells(S) = \bigcup_{i=1}^{n} f_i$ and $\forall i, j$ where $i \neq j : f_i \cap f_j = \emptyset$. We call $F = \{f_1, \ldots, f_n\}$ a complete fragmentation of a spreadsheet $S$.*

*Example 5.* A possible complete fragmentation for the spreadsheet in Fig. 1 is $F = \{\{\text{C2, D2, E2}\}, \{\text{B5, C5, D5}\}, \{\text{B6, C6, D6}\}\}$. Another complete fragmentation is $F' = \{\{B5, B6\}, \{C2, C5, C6\}, \{D2, D5, D6\}, \{E2\}\}$. Even a fragmentation containing only a single fragment ($F'' = \{\{\text{C2, D2, E2, B5, C5, D5, B6,C6, D6}\}\}$) is a complete and therefore valid fragmentation.

**Collapsing.** We start the fragmentation process by collapsing copy-equivalent cells which share the same column or row index. The idea behind this is that we want to avoid to collapse cells which appear somewhere else in the spreadsheet and have the same formula only by chance. To do so, we first define a fragment that is column-row-related, i.e., that only contains cells which share the same column or row with another cell of the fragment.

**Definition 10 (Column-Row-Related).** *A fragment $f$ is column-row-related, if a graph can be spanned over all cells $c \in f$ with nodes $f$ and edges $e$ such that the graph connects all nodes in $f$ and $\forall (c, c') \in e : x(c) = x(c') \vee y(c) = y(c')$.*

*Example 6.* In our running example, the fragment $f = \{\text{D2}, \text{B5}\}$ is not column-row-related, as the cells D2 and B5 do not share the same column or row. The fragment $f' = \{\text{D2}, \text{B5}, \text{D5}\}$, however, is colum-row-related, because cell D5 shares the same column with D2 and the same row with B5.

With this definition, we can now define base fragments that represent the collapsed formulas. They comprise either a single formula or a set of copy-equivalent formulas. The base fragments are later used in the merging step.

**Definition 11 (Base Fragment).** *We call a fragment a base fragment if all contained cells are copy-equivalent and if they share the same row or column with another cell. More formally, a fragment $f$ is a base fragment if $\forall c, c' \in f$ : copy-equivalent$(c, c') \wedge$ column-row-related$(f) \wedge \nexists$ base fragment $f'$ with $f \subset f'$. The left-most cell in the first row of a base fragment $f$ is called the representative of $f$. The function representative$(f)$ returns a cell $c \in f$ as the representative such that $\forall c' \in f : y(c) < y(c') \vee (y(c) = y(c') \wedge x(c) \leq x(c'))$.*

Collapsing copy-equivalent cells has the benefit that the complexity of test cases can be reduced. Instead of indicating input data for each individual input cell, the user has to indicate only the input data required for one cell (the representative) of a set of copy-equivalent cells.

*Example 7.* In our example, the copy-equivalent cells C2, D2, and E2 have the same column index and can, therefore, be collapsed. A test case for the fragment {C2, D2, E2} only requires values for the input cells B2, B3, and B4 and the output cell C2. No values have to be specified for the cells referenced in D2 and E2 as well as for the cells D2 and E2 themselves. In total, our running example has five base fragments: {C2, D2, E2}, {B5, C5, D5}, {B6}, {C6}, and {D6}.

**Merging.** For the second part of the fragmentation process, i.e., merging the base fragments, the goal is to find the optimal fragmentation based on some complexity criteria. Because of the number of possible solutions, deterministically finding the optimal solution is not possible for larger spreadsheets. Therefore we use an evolutionary algorithm. Evolutionary algorithms follow two concepts from biology: evolution and selection, i.e., survival of the fittest. We implement the evolution process by randomly merging fragments. We use randomness to imitate biologic evolution; some of the newly created fragments are nearer to an optimal fragmentation, others are far away.

**Definition 12 (Mergeable).** *Two base fragments $f, f'$ can be merged if $|f| = |f'| \wedge \forall c \in f \ \exists c' \in f' : (x(c) - x(r) = x(c') - x(r')) \wedge (y(c) - y(r) = y(c') - y(r'))$ where $r =$ representative$(f)$ and $r' =$ representative$(f')$.*

The result of merging two fragments is a fragment that contains the cells of both fragments. As the base fragments of a spreadsheet are defined in a unique way, a merged fragment can always be partitioned into its base fragments again. Therefore, we can generalize the merging process of arbitrary fragments as follows: two arbitrary fragments consisting of one or more base fragments can be merged if all the base fragments that they comprise can be merged.

*Example 8.* The base fragments {C2, D2, E2} and {B5, C5, D5} can be merged because the copy-equivalent cells of these fragments have the same distance to

their representatives (C5/D5 is one/two column(s) left of B5; D2/E2 is one/two column(s) left of C2). {B6}, {C6}, and {D6} can be merged because they do not comprise any copy-equivalent cells. The result is the fragmentation {{C2, D2, E2, B5, C5, D5}, {B6, C6, D6}}. Both fragments are not base fragments as not all contained cells are copy-equivalent. Merging all combinations of base fragments is not possible, e.g., it would not be possible to merge the base fragments {C2, D2, E2} and {B6}, because the size of these two base fragments is different.

In the genetic process, we randomly test different fragmentations. These fragmentations are called mutants (or individuals). Several mutants build a population which evolves from generation to generation. Only the fittest mutants survive their generation. In each generation, the population is extended with newly generated mutants. The goal of this process is to find the fragmentation that leads to the most "useful" fragments, i.e., a well-structured and comprehensible partition of the spreadsheet. We measure the usefulness of a given fragmentation by means of the aggregated complexity of its fragments. To determine the complexity of each fragment, we use several heuristics: the number of input and output values, the spanned area of the fragment, and the complexity of the formulas. These heuristics are based on the ideas of code smells [11] and spreadsheet complexity measures [12]. Instead of considering all cells of the fragments in the heuristics, we consider only their representatives.

**Definition 13 (Representatives).** *The function representatives(f) for a fragment $f$ which consists of $n$ base fragments ($f_{b1}, \ldots, f_{bn}$) returns the set of representatives of the base fragments:*

$$representatives(f) = \bigcup_{f_{bi} \in f} representative(f_{bi}).$$

The heuristics are defined as follows.

$$H_{in}(f) = |input(r)| \tag{1}$$

$$H_{out}(f) = |output(r)| \tag{2}$$

$$H_{area}(f) = (max_x(r) - min_x(r) + 1) * (max_y(r) - min_y(r) + 1) \tag{3}$$

$$H_{formulas}(f) = \sum_{c \in r} formulaComplexity(c) \tag{4}$$

where $r = representatives(f)$, $max_x$ returns the largest value for $x$ for a set of cells, $min_x$ returns the smallest value, and the *formulaComplexity* of a cell is measured by the number of conditionals and cell references in the formula.

Heuristic (1) aims to group cells which have the same input and the idea of (2) is to minimize the number of output cells. Heuristic (3) favors fragmentations that contain "physically" close cells over fragmentations that comprise distant cells. Heuristic (4) sums up the formula complexities of all representative cells to compute the formula complexity of a fragment.

*Example 9.* For the fragmentation $F = \{f_1 = \{C2, D2, E2\}, f_2 = \{B5, C5, D5\}, f_3 = \{B6, C6, D6\}\}$ of our running example, we have the following values:

| Fragment | $f_1$ | $f_2$ | $f_3$ |
|---|---|---|---|
| $H_{in}$ | 3 | 3 | 3 |
| $H_{out}$ | 1 | 1 | 3 |
| $H_{area}$ | 1 | 1 | 3 |
| $H_{formulas}$ | 3 | 5 | 6 |

These four heuristics are used to determine the fitness of the individuals. Each heuristic results in a single number for each fragment. The heuristics vary in their importance for determining a good fragmentation. Therefore, we weight the different numbers before we sum them up for each fragment.

$$\mathit{fragmentComplexity}(f) = \sum_{i=0}^{|H|} H_i(f) * w_i \tag{5}$$

where $H$ is the list of all implemented heuristics and $w$ is a vector containing the weights of the individual heuristics. The weights can be set manually or with the help of some optimization technique. The fragment complexities are then used to determine the fitness of the fragmentation as a whole. To support a balanced fragmentation in which all fragments have roughly the same complexity, we also take the standard deviation of all fragment complexities into account.

$$\mathit{fitness}(F) = -\left(\sum_{f \in F} \mathit{fragmentComplexity}(f)\right) - \sigma(F) * w_\sigma \tag{6}$$

where $\sigma(F)$ is the standard deviation and $w_\sigma$ is its weight:

$$\sigma(F) = \sqrt{\frac{\sum_{f \in F} \left(\mathit{fragmentComplexity}(f) - \frac{\sum_{f \in F} \mathit{fragmentComplexity}(f)}{|F|}\right)^2}{|F|}}.$$

The resulting number represents the fitness of the fragmentation, i.e., the inverse complexity. The higher the number, the less complex is the fragmentation. We aim to find the individual that has the lowest complexity.

*Example 10.* Assume we have the weighting vector $w = (2, 3, 4, 5)$ and $w_\sigma = 2$. The complexities for the fragments of Example 9 are $\mathit{fragmentComplexity}(f_1) = 3{*}2{+}1{*}3{+}1{*}4{+}3{*}5 = 28$, $\mathit{fragmentComplexity}(f_2) = 3{*}2{+}1{*}3{+}1{*}4{+}5{*}5 = 38$, $\mathit{fragmentComplexity}(f_3) = 3 * 2 + 3 * 3 + 3 * 4 + 6 * 5 = 57$. Then $\mathit{fitness}(F) = -(28 + 38 + 57) - \sqrt{\frac{(28-41)^2+(38-41)^2+(57-41)^2}{3}} * 2 = -147.1$.

Algorithm 1 illustrates the fragment generation process. As an input the algorithm takes the set $S$ of formula cells of the spreadsheet that should be fragmented, the population size $p$, i.e., the number of individuals that can exist at any time, the number of generations $g$, the percentage $s$ of mutants that should survive in the population in each generation, and the heuristic weights $w$ and $w_\sigma$. The procedure *CollapseCopyEquivalentCells(S)* (Line 2) takes as input

**Algorithm 1** Fragment Generation

```
1: procedure GenerateFragments(S, p, g, s, w, w_σ)        ▷ S... set of cells
                                  ▷ p... population        ▷ g... #generations
                          ▷ s... survival rate             ▷ w, w_σ... heuristic weights
2:     B ← CollapseCopyEquivalentCells(S)
3:     count ← 0
4:     P ← createInitialPopulation(B, p)
5:     while count < g do
6:         P ← selectFittestIndividuals(P, s, w, w_σ)
7:         P ← P ∪ getMutants(P, p − |P|)
8:         count ← count + 1
9:     end while
10:    F ← selectFittestIndividual(P, w, w_σ)
11:    return F
12: end procedure
```

a set of cells and creates the base fragments according to Definition 11. The resulting fragmentation $B$ is stored as the base fragmentation and is also used to create the initial population (Line 4).

In the evolution step, the fittest individuals are selected. The function *selectFittestIndividuals* takes as input the population $P$, the selection rate $s$, and the heuristic weights $w$ and $w_\sigma$ and computes the fitness value for each individual $F \in P$ according to (6). The $s * p$ individuals with the highest fitness are kept in the population. In Line 7 mutants are created until the number of individuals is equal to the population size $p$. The mutants are generated by randomly merging fragments or dividing them into their base fragments. All mergings are done with respect to Definition 12. In the next generations, evolution and selection repeat. In Line 10, the fittest mutant, i.e., the individual with the lowest complexity value and standard deviation, is returned.

### 4.2 Fragment-Based Diagnosis

The idea of fragment-based diagnosis is to efficiently locate the formulas that can be the cause of unexpected outcomes. To do so, we use the generated fragments as the smallest diagnosable components in the diagnosis process and reformulate the diagnosis problem accordingly. First, we set the diagnosable components COMPS as the generated fragments in $F$, i.e., $\text{COMPS} = F$. Then, we reformulate the system description SD so that the abnormal predicates use the fragment the corresponding formula cells belong to.

*Example 11.* In our example we reformulate the system description as $\text{SD} = \{\text{AB}(f_1) \vee \text{C2} = \text{B2} + \text{B3} * \text{B4}, \text{AB}(f_1) \vee \text{D2} = \text{C2} + \text{C3} * \text{C4}, \ldots, \text{AB}(f_3) \vee \text{D6} = \text{B5} + \text{C5} + \text{D5}\}$ and $\text{COMPS} = \{f_1, f_2, f_3\}$.

The reformulation of SD ensures that once a fragment is considered to be incorrect in the MBD process, all the formulas of this fragment are also considered

to be incorrect when searching for inconsistencies between the expected and observed calculation outcomes. With the help of this formulation, the complexity of the diagnosis process can be strongly reduced, as only entire fragments can be considered to be incorrect.

Overall, we can now automatically generate a fragmentation of a spreadsheet and then determine those fragments that can be the cause of the erroneous outcome. The user can then ignore all fragments (and their cells) that are not part of any diagnosis. For the remaining fragments, the user can inspect their formulas to manually find the fault or he/she can specify additional test cases for these fragments. To specify a test case for a fragment, the user can choose the values for the input cells of the fragment (regardless of whether these cells contained input values in the original spreadsheet or formulas) and then state the expected values for any of the output or interim cells of the fragment. These additional simple test cases help to reduce the number of diagnoses in the MBD process and therefore make it easier to find the true cause of the problem. If the user for example specifies complete test cases containing expected values for all interim and output cells for the fragments that contain the actually faulty formulas, the true cause of the problem will always be found as the only diagnosis.

## 5 Empirical Evaluation

In this section, we first describe the framework for evaluating our approach and the characteristics of the evaluated spreadsheets. Afterwards, we present and discuss the results of our initial empirical evaluation.

We integrated the proposed approach into the EXQUISITE Framework [15, 17]. The back-end used for calculating the fragmentation and the diagnoses was implemented in Java. We used JGAP to implement the genetic approach and Choco as the constraint solver for the MBD process. After preliminary tests we empirically set the weight vector for the fragmentation process to $w = (0.2, 1, 1, 1)$ and $w_\sigma = 0.2$ to obtain useful fragmentations. The experiments were run on a laptop computer with an Intel Core i7-4710MQ CPU running Windows 8.1.

We evaluated our approach on 5 different spreadsheets. The characteristics of these spreadsheets are shown in Table 1. The tested spreadsheets were quite diverse with regard to the number of formula cells, which range from 9 to 457. However, as it is common for most real-world spreadsheets [10], the number of unique formulas is much lower than the number of formulas itself. One spreadsheet contains a single faulty formula. The *Proteins* spreadsheet contains two unique faults. The other spreadsheets have a fault that was copied to other cells.

Table 2 shows the time needed for the fragmentation, the number of fragments that were generated, the number of cells that could be collapsed due to their copy-equivalence, and the number of merged cells to form larger fragments. The time needed for the fragmentation process mainly depends on the number of fragments that can be merged after the collapsing step. The fragmentation process was therefore faster for larger spreadsheets with many copy-equivalent cells (e.g. *Course planning*) than for smaller spreadsheets with many different

**Table 1.** Characteristics of the tested spreadsheets.

| **Spreadsheet** | #Input cells | #Formula cells | #Unique formulas | #Faults |
|---|---|---|---|---|
| Wage planning | 69 | 63 | 25 | 1 |
| Proteins | 14 | 98 | 14 | 2 |
| Sales forecast | 224 | 143 | 4 | 2 |
| Course planning | 126 | 457 | 10 | 2 |
| Velocity calculation | 7 | 9 | 5 | 3 |

**Table 2.** Results of the fragmentation process.

| **Spreadsheet** | Time [ms] | #Fragments | #Collapsed cells | #Merged cells |
|---|---|---|---|---|
| Wage planning | 151 | 13 | 38 | 12 |
| Proteins | 55 | 4 | 84 | 10 |
| Sales forecast | 26 | 4 | 139 | 0 |
| Course planning | 49 | 4 | 447 | 6 |
| Velocity calculation | 10 | 3 | 4 | 2 |

formulas (e.g. *Wage planning*). Overall, most of the tested spreadsheets had many copy-equivalent cells that could be collapsed in the fragmentation process so that the merging step could be completed very fast. Regarding the number of generated fragments the second smallest spreadsheet (*Wage planning*) led to the largest number of fragments, as it had the highest number of unique formulas.

Table 3 shows the results of calculating the diagnoses with the different approaches. Calculating the diagnoses based on the fragments was faster for all tested spreadsheets than calculating the diagnoses based on the individual cells. Even for the most complex *Course planning* spreadsheet for which we needed almost 12 seconds to compute the cell-based diagnoses, we were able to compute the fragments that could be the cause of the error in about 14 milliseconds. Regarding the number of diagnoses, the fragment-based diagnosis procedure determined about half of the fragments as possible diagnoses for the observed error. This is also promising, as the reduced number of fragments to inspect lets the user focus on those parts of the spreadsheet that can be the real cause of the error while the others can be ignored.

The columns "Cell-based using additional test cases" show the results of diagnosing the spreadsheets on the cell level, but with additional test cases for the generated fragments. For this measurement we simulated a user and manually created test cases by specifying values for all interim and output cells of those fragments that were part of a diagnosis. With this additional information, we were always able to exactly locate the true cause of the observed errors. For those spreadsheets for which the original cell-based diagnosis needed more than 100 milliseconds, our approach was also much faster, even if we add up all calculation times in this process (last column of Table 3), i.e., the time to generate the fragments (see Table 2), the time to calculate the possibly faulty fragments, and the time to determine the true cause of the error with the additional test cases.

**Table 3.** Results of the different diagnosis procedures. Times are given in milliseconds.

| Spreadsheet | Cell-based | | Fragment-based | | Cell-based using additional test cases | | Overall |
|---|---|---|---|---|---|---|---|
| | Time | #Diag | Time | #Diag | Time | #Diag | Time |
| Wage planning | 14 | 24 | 3 | 6 | 15 | 1 | 169 |
| Proteins | 440 | 85 | 11 | 2 | 126 | 1 | 192 |
| Sales forecast | 102 | 144 | 3 | 2 | 48 | 1 | 77 |
| Course planning | 11,903 | 2,304 | 14 | 2 | 1,900 | 1 | 1,963 |
| Velocity calculation | 4 | 11 | 0 | 2 | 3 | 1 | 13 |

In cases in which the fragmentation process requires more time than the cell-based diagnosis process, our fragment-based approach cannot help in terms of the overall computation time. However, our approach is still helpful to reduce the number of diagnosis candidates in these cases.

## 6 Related Work

Reiter has laid the groundwork for modern Model-Based Diagnosis (MBD) approaches with his theory about diagnosis reasoning from first principles [20]. In the last decades, researchers applied his concept to different areas of application, e.g., logic programs [4] and hardware design languages [8]. Due to the high computational complexity of "pure" MBD, various researchers have explored the consideration of abstractions in the process. There are two main types of abstraction: behavioral abstraction and structural abstraction. Behavioral abstraction simplifies the description of the model; structural abstraction aggregates components of the model. The approach presented in this paper belongs to the group of structural abstractions. Autio and Reiter were among the first proposing structural abstractions [2]. Other relevant works on structural abstraction include Chittaro and Ranon's work on hierarchical MBD in general [3], Stumptner and Wotawa's work on diagnosing tree-structured systems [22], and Felfernig *et al.*'s work on structural abstraction to debug configurator knowledge bases [7].

Our MBD approach directly builds upon the consistency-based approach presented in [17], in which diagnoses are computed indirectly via conflict sets. In contrast, the consistency-based approach presented in [1] computes the diagnoses directly by means of an SMT solver. To the best of our knowledge, structural abstractions have not yet been applied to MBD-based spreadsheet debugging techniques. However, Hofer and Wotawa have recently proposed dependency-based models for spreadsheets as behavioral abstractions [14].

Cunha *et al.* proposed a system to automatically infer a model from a spreadsheet [5]. Their approach is based on the values of the cells to decide on the structure of the spreadsheet, while ours uses the formulas. In general, spreadsheet debugging is a sub-field of spreadsheet quality assurance (QA), which also covers testing, static analysis, modeling, visualization, design, and maintenance support. An overview of QA techniques for spreadsheets can be found in [18].

## 7 Conclusion

Through our empirical evaluation we could demonstrate that partitioning spreadsheets into fragments can be a powerful means to improve MBD for spreadsheets. On the one hand, the number of diagnoses is decreased; on the other hand, the time required for computing the diagnoses is significantly reduced.

Furthermore, users can benefit from fragments when testing spreadsheets in additional ways. For large spreadsheets, it can be difficult to determine if the spreadsheet computes the correct output values for the given input values. Fragments are units that can be tested separately and a user can create several test cases for each fragment. Because of the small size of the fragments, a user can easily determine the correctness of computed values. In addition, it is easier for the user to manually specify values that cover special cases (e.g., conditionals, division by zero) for small fragments than for a whole spreadsheet. Testing on the fragment level can be compared to unit testing in software.

A number of research questions exist which we plan to answer in future work. First, we plan to evaluate our approach by means of a user study. This user study should settle the question of the usability of our approach for end users. Additionally, we plan to expand our approach to automatically create input values for test cases on the fragment level. These input values should help to test the "standard" behavior of the spreadsheet as well as the special cases. In a next step, we will evaluate whether users prefer to use these automatically generated fragments for testing over manually created ones.

The fragmentation process itself can be improved in several ways. Up to now, we have used fixed values for the population size and the weights of the heuristics. In our future research, we plan to examine whether or not we can obtain a better fragmentation when we change the values of these parameters. In addition, we plan to add additional heuristics to the fitness function. A further speed-up or a better quality of the fragment computation can be achieved when changing the algorithm termination criteria from a fixed number of generations to another criterion. For example, we could stop the evolutionary process when the fitness remains constant over several generations. Once these improvements are implemented, we will evaluate the approach on a larger set of spreadsheets.

## Acknowledgment

The work described in this paper was funded by the Austrian Science Fund (FWF) under contract number I2144 and the German Research Foundation (DFG) under contract number JA 2095/4-1.

## References

1. Abreu, R., Hofer, B., Perez, A., Wotawa, F.: Using constraints to diagnose faulty spreadsheets. Software Quality Journal 23(2), 297–322 (2015)

2. Autio, K., Reiter, R.: Structural Abstraction in Model-Based Diagnosis. In: ECAI '98. pp. 269–273 (1998)
3. Chittaro, L., Ranon, R.: Hierarchical model-based diagnosis based on structural abstraction. Artificial Intelligence 155(1-2), 147–182 (2004)
4. Console, L., Friedrich, G., Dupré, D.T.: Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. In: AADEBUG '93. pp. 85–87 (1993)
5. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: VL/HCC '10. pp. 93–100 (2010)
6. F1F9: The Dirty Dozen, http://blogs.mazars.com/the-model-auditor/files/2014/01/12-Modelling-Horror-Stories-and-Spreadsheet-Disasters-Mazars-UK.pdf, last visited: April, 7th 2016
7. Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., Zanker, M.: Hierarchical Diagnosis of Large Configurator Knowledge Bases. In: KI: Advances in Artificial Intelligence. pp. 185–197 (2001)
8. Friedrich, G., Stumptner, M., Wotawa, F.: Model-based diagnosis of hardware designs. Artificial Intelligence 111(1-2), 3–39 (1999)
9. Hermans, F., Pinzger, M., van Deursen, A.: Supporting professional spreadsheet users by generating leveled dataflow diagrams. In: ICSE '11. pp. 451–460 (2011)
10. Hermans, F., Murphy-Hill, E.R.: Enron's Spreadsheets and Related Emails: A Dataset and Analysis. In: ICSE '15. pp. 7–16 (2015)
11. Hermans, F., Pinzger, M., van Deursen, A.: Detecting code smells in spreadsheet formulas. In: ICSM '12. pp. 409–418 (2012)
12. Hodnigg, K., Mittermeir, R.T.: Metrics-Based Spreadsheet Visualization - Support for Focused Maintenance. In: EuSpRIG '08. pp. 79–94 (2008)
13. Hofer, B., Riboira, A., Wotawa, F., Abreu, R., Getzner, E.: On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets. In: FASE '13. pp. 68–82 (2013)
14. Hofer, B., Wotawa, F.: Why does my spreadsheet compute wrong values? In: ISSRE '14. pp. 112–121 (2014)
15. Jannach, D., Baharloo, A., Williamson, D.: Toward an integrated framework for declarative and interactive spreadsheet debugging. In: ENASE '13. pp. 117–124 (2013)
16. Jannach, D., Schmitz, T.: Using calculation fragments for spreadsheet testing and debugging. In: SEMS '15. pp. 1–2 (2015)
17. Jannach, D., Schmitz, T.: Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. Automated Software Engineering 23(1), 105–144 (2016)
18. Jannach, D., Schmitz, T., Hofer, B., Wotawa, F.: Avoiding, finding and fixing spreadsheet errors - A survey of automated approaches for spreadsheet QA. Journal of Systems and Software 94, 129–150 (2014)
19. Junker, U.: QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In: AAAI '04. pp. 167–172 (2004)
20. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1), 57–95 (1987)
21. Schmitz, T., Jannach, D.: Finding Errors in the Enron Spreadsheet Corpus. In: VL/HCC '16 (2016)
22. Stumptner, M., Wotawa, F.: Diagnosing tree-structured systems. Artificial Intelligence 127(1), 1–29 (2001)
23. Tan, G.: Spreadsheet mistake costs Tibco shareholders $100 million, http://on.wsj.com/1vjYdWE, published: 2014-10-16; last visited: 2016-04-07