Testing for Distinguishing Repair Candidates in Spreadsheets – the Mussco Approach

Rui Abreu, Simon Ausserlechner, Birgit Hofer, Franz Wotawa

Testing for distinguishing repair candidates in spreadsheets – the Mussco approach

Rui Abreu[1], Simon Außerlechner[2], Birgit Hofer[2], and Franz Wotawa[2]

[1] Palo Alto Research Center
Palo Alto, CA USA
Email: rui@computer.org
[2] Graz University of Technology Graz, Austria
Email: {bhofer,wotawa}@ist.tugraz.at

Abstract. Companies and other organizations use spreadsheets regularly as basis for evaluation or decision-making. Hence, spreadsheets have a huge economical and societal impact and fault detection, localization, and correction in the domain of spreadsheet development and maintenance becomes more and more important. In this paper, we focus on supporting fault localization and correction given the spreadsheet and information about the expected cell values, which are in contradiction with the computed values. In particular, we present a constraint approach that computes potential root causes for observed behavioral deviations and also provide possible fixes. In our approach we compute possible fixes using spreadsheet mutation operators applied to the cells' equations. As the number of fixes can be large, we automatically generate distinguishing test cases to eliminate those fixes that are invalid corrections. In addition, we discuss the first results of an empirical evaluation based on a publicly available spreadsheet corpus. The approach generates on average 3.1 distinguishing test cases and reports 3.2 mutants as possible fixes.

Keywords: fault localization, spreadsheet debugging, distinguishing test-cases, spreadsheet mutations.

1 Introduction

Spreadsheets are a flexible end-users programming environment. "End-user" programmers vastly outnumber professional ones: the US Bureau of Labor and Statistics estimates that more than 55 million people used spreadsheets and databases at work on a daily basis by 2012 [14]. 95 % of all U.S. companies use spreadsheets for financial reporting [19], and 50 % of all spreadsheets are the basis for decisions.

Numerous studies have shown that existing spreadsheets contain redundancies and errors at an alarmingly high rate, e.g., [6]. This high error rate can be explained with the lack of fundamental support for testing, debugging, and structured programming in the spreadsheet world. Errors in spreadsheets may entail a serious economical impact, causing yearly losses worth around 10 billion

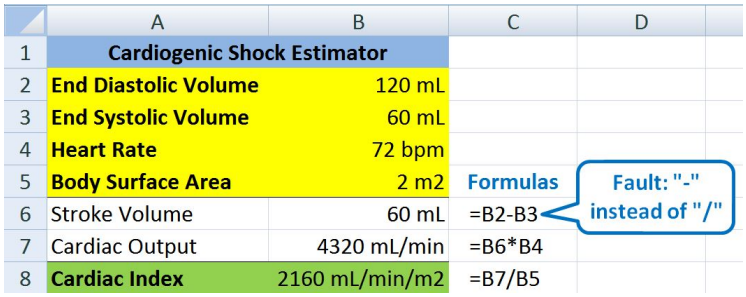


Fig. 1: The Cardiogenic shock estimator spreadsheet

dollars [18][3]. This paper improves the state-of-the-art in spreadsheet debugging by proposing an approach for correcting faults in spreadsheets.

In this paper, we make use of the running example illustrated in Fig. 1. This spreadsheet is used by physicians to estimate cardiogenic shock[4]. Cells `B2` to `B5` are those cells that need an input from the user. Cell `B8` shows the result of the computation from which physicians derive their conclusions. Cell `B6` is faulty. It computes `B2/B3` instead of `B2-B3`. As a consequence, the value of cell `B8` is outside the bounds even when the patient's input values are okay. If the physician notices that the computed value is outside the bounds, he might want to *debug* the spreadsheet.

In this paper, we use constraint-based techniques for spreadsheet debugging [13, 3]. These techniques take as input a faulty spreadsheet and a test case[5] that reveals the fault in order to compute a set of diagnosis candidates (cells). The spreadsheet and the test case are converted into a constraint satisfaction problem (CSP). A constraint or SMT (satisfiability modulo theories) solver is used to obtain the set of diagnosis candidates. A major limitation of these approaches is that they yield many diagnosis candidates. To avoid this problem, we propose to integrate testing for restricting the number of diagnosis candidates. In particular, we propose to compute possible corrections of the program (using mutation techniques) and from these distinguishing test cases. A test case is a distinguishing test case if and only if there is at least one output variable where the computed value of two mutated versions of a spreadsheet differ on the same input. We have two main contributions : (1) We propose Mussco (Mutation Supported Spreadsheet COrrection), an approach to fault localization in spreadsheets that relies on constraint-based reasoning to provide suggestions for possible fixes by applying spreadsheet mutation operators. Since the number of such mutants can be large, our approach automatically generates distinguishing test cases to eliminate mutants that are invalid corrections. (2) We carried out an empirical evaluation using the publicly available Integer Spreadsheet Corpus.

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

[4] A cardiogenic shock is when the heart has been damaged so much that it is unable to supply enough blood to the organs.

[5] A test case specifies values for the input cells as well as the expected values for the output cells.

Results show that on average 3.1 distinguishing test cases are generated and 3.2 mutants are reported as possible fixes. On average, generating mutants and distinguishing test cases requires 47.9 seconds, rendering the approach applicable as a real-time application.

2 Basic Definitions

In this paper, we rely on the spreadsheet language $\mathcal{L}$ defined by Hofer *et al.* [11]. We refer the interested reader to that paper for more information about the syntax and semantics of the underlying spreadsheet language. For the sake of completeness, we state the most important concepts and definitions in the following paragraphs.

Every spreadsheet is a matrix of cells that are uniquely identifiable using their corresponding column and row number. The function φ maps the cell names from a set $CELLS$ to their corresponding position (x, y) in the matrix where x represents the column and y the row number. The functions φ_x and φ_y return the column and row number of a cell respectively. Each cell $c \in CELLS$ has a corresponding value $\nu(c)$ and an expression $\ell(c)$. The value of a cell can be either undefined ϵ, an error $\perp$, or any number, Boolean or string value. The expression of a cell $\ell(c)$ can either be empty or an expression written in the language $\mathcal{L}$. The value of a cell c is determined by its expression. If no expression is explicitly declared for a cell, the function ℓ returns ϵ while the function ν returns 0.

An area $c_1 \underline{:} c_2 \subseteq CELLS$ is a set consisting of all cells that are within the area spanned by the cells c_1, c_2 , i.e.:

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

For our debugging approach, we require information about cells that occur in an expression, i.e. the referenced cells. The function $\rho : \mathcal{L} \mapsto 2^{CELLS}$ returns the set of referenced cells.

Definition 1 (Spreadsheet). *A countable set of cells $\Pi \subseteq CELLS$ is a spreadsheet if all cells in Π have a non-empty corresponding expression or are referenced in an expression, i.e., $\forall c \in \Pi : (\ell(c) \neq \epsilon) \vee (\exists c' \in \Pi : c \in \rho(\ell(c')))$.*

This definition restricts spreadsheets to be finite. For our approach, we only consider loop-free spreadsheets, i.e., spreadsheets that do not contain cycles within the computation. Therefore, we introduce the notation of data dependence between cells, and the data dependence graph.

Definition 2 (Direct dependence). *Let c_1, c_2 be cells of a spreadsheet Π. The cell c_2 directly depends on cell c_1 if and only if c_1 is used in c_2's corresponding expression, i.e., $dd(c_1, c_2) \leftrightarrow (c_1 \in \rho(\ell(c_2)))$.*

Definition 3 (Data dependence graph). *The data dependence graph (DDG) of a spreadsheet Π is a tuple (V, A) with V being a set of vertices comprising*

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

From this definition, we are able to define general dependence between cells. Two cells of a spreadsheet are dependent if and only if there exists a path between the corresponding vertices in the DDG. A spreadsheet Π is feasible if and only if its DDG is acyclic. From here on, we assume that all spreadsheets we consider for debugging are feasible. Hence, we use the terms spreadsheet and feasible spreadsheet synonymously. For debugging, we have to define test cases for distinguishing faulty spreadsheets from correct spreadsheets.

Definition 4 (Input, output). *Given a feasible spreadsheet Π and its DDG (V, A), then the input cells of Π (or short: inputs) comprise all cells that have no incoming edges in the corresponding vertex of Π's DDG. The output cells of Π (or short: outputs) comprise all cells where the corresponding vertex of the DDG has no outgoing vertex.*

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

All formula cells of a spreadsheet that serve neither as input nor as output are called intermediate cells. With the definition of inputs and outputs, we can now define test cases.

Definition 5 (Test case). *A test case T for a spreadsheet Π is a tuple (I, O) where I is a set of pairs (c, v) specifying the values for all $c \in inputs(\Pi)$ and O is a set of pairs (c, e) specifying the expected values for some output cells. T is a failing test case for spreadsheet Π if there exists at least one cell c where the expected value e differs from the computed value $\nu(c)$ when using I on Π.*

We say that a test case is a passing test case if all computed values are equivalent to the expected values.

Definition 6 (Spreadsheet debugging problem). *A spreadsheet Π and a failing test case T form a spreadsheet debugging problem.*

Example 1. The test case T with $I = \{(\texttt{B2}, 120), (\texttt{B3}, 60), (\texttt{B4}, 72), (\texttt{B5}, 2)\}$ and $O = \{(\texttt{B8}, 2160)\}$ is a failing test case for the Cardiogenic shock estimator spreadsheet. This test case together with the spreadsheet forms a debugging problem.

A solution of a spreadsheet debugging problem (Π, T) is a set of cells that explain the faulty behavior. In particular, we say that an explanation Π^E is itself a spreadsheet comprising the same cells as Π but different cell expressions that make the test case T a passing test case for Π^E.

Example 2. A spreadsheet Π_1 where the expression of cell `B6` is changed to `B2 - B3` is obviously an explanation that makes the test case T a passing one. However, a spreadsheet Π_2 where we change the expression of `B7` to `30 * B6 * B4` is an explanation as well.

3 Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) is a tuple (V, D, C) where V is a set of variables with a corresponding domain from D, and C is a set of constraints [8]. Each constraint has a set of variables and specifies the relation between the variables. Abreu *et al.* [3] have shown how to state the spreadsheet debugging problem as a CSP. To be self-contained, we briefly explain the conversion in Algorithm Convert (Fig. 2). Details about the conversion can be found in the work of Abreu *et al.* [3]. Formula cells are concatenated with a variable representing the health state of that formula: A cell c is faulty, i.e. c does not behave as expected, or the constraints representing the formula must be satisfied (Line 3). The expressions of the formula cells are converted using the Algorithm ConvExp which works as follows: Constants are represented by themselves. Cell references are mapped to the corresponding variables. In case of compound expressions, the conversions of the single expressions and the constraint representing the compound expression are added to the constraint system. The values of the input cells and the expected values indicated in the test case are added to the constraint system (Line 6).

Input: Spreadsheet Π, test case $T = (I, O)$
Output: A set of constraints representing Π and T.
1: CONS $= \emptyset$
2: **for** $c \in (\Pi \setminus \mathit{inputs}(\Pi))$ **do**
3: CONS $=$ CONS $\cup \{ab_c \vee v_c == \mathsf{ConvExp}(\ell(c))\}$
4: **end for**
5: **for** tuples $(\mathit{cell}, \mathit{value}) \in (I \cup O)$ **do**
6: CONS $=$ CONS $\cup \{v_{cell} == \mathit{value}\}$
7: **end for**
8: **return** CONS

Fig. 2: Algorithm $\mathsf{Convert}(\Pi, T)$

Example 3. The constraint representation of our example from Fig. 1 is: $B2 == 120$, $B3 == 60$, $B4 == 72$, $B5 == 2$, $ab_{B6} \vee B6 = B2/B3$, $ab_{B7} \vee B7 = B6{*}B4$, $ab_{B8} \vee B8 = B7/B5$, $B8 == 2160$.

Since spreadsheets must be finite, the Convert algorithm terminates. The computational complexity of the algorithm is $O(|CELLS| \cdot L)$ where L is the maximum length of an expression. For computing diagnoses, let SD be the obtained constraint representation for a spreadsheet Π. A diagnosis Δ is a subset of the cells contained in Π such that SD $\cup \{\neg ab_c | c \in \Pi \setminus \Delta\} \cup \{ab_c | c \in \Delta\}$ is satisfiable. We use an SMT solver for computing solutions for a given CSP. The theoretic background of using SMT solvers for CSPs is explained by Liffiton and Sakallah [15, 16].

4 Mutation creation

With the previously described fault localization technique, the user only gets the information which cells have to be changed, but not how the cells have to be changed. We are confident that the information of how cells have to be changed is important for the user. Therefore, we propose an approach that automatically creates versions, i.e. mutants, of the spreadsheet that satisfy the given test case.

Weimer *et al.* [23] introduced genetic programming for repairing C programs. Similar to them, we make assumptions how to restrict the search space. For example, we perform mutations on the cone for a given cell[6] and Weimer *et al.* make mutations on the weighted path. In addition, Weimer *et al.* assume that the programmer has written the correct statement somewhere else in the program. We assume that when a spreadsheet programmer referenced the wrong cell, the correct cell is in the surrounding of the referenced cell. However, we differ from their genetic programming approach as we do not use crossover and randomness for selecting mutations.

A primitive way to compute mutants is to clone the spreadsheet and change arbitrary operators and operands in all formulas of the cells contained in one diagnosis. If the created mutant satisfies the given test case we present the mutant to the user. Otherwise we discard the mutant and create another mutant. The problem with this approach is that too many mutants have to be computed until the first mutant passes the given test case. Therefore, we propose a more sophisticated approach which includes the mutation creation process in the CSP. Instead of only transforming cell formulas into a value-based constraint model, we also include the information how the cells could be mutated. We allow the following mutation operations:

- replace constant with reference or other constant
- replace reference with constant or other reference
- replace arithmetical (relational) operator with other arithmetical (relational) operator
- replace function with other functions of the same arity
- resize areas

We are aware that these mutation operators are not able to correct all faulty spreadsheets. In particular, the creation of completely new formulas is up to future work.

When creating mutants, we have to face two challenges: (1) The created mutant must be a feasible spreadsheet. (2) Theoretically, an infinite number of mutations can be created. To handle the first challenge, we propose the following solution: Each cell that is represented in the CSP gets an additional Integer variable with the domain $\{1, |\Pi|\}$. The constraint solver has to assign values to these variables in such a way that each cell gets a number that is higher than the numbers assigned to the cells this cell references. This constraint ensures that

[6] A cone for a cell c is recursively defined as the union of all cones of the cells which are referenced in c and the cell c itself.

the created mutant is still a feasible spreadsheet. To handle the second challenge, we reduce the search space by making following restrictions:

- Mutations are only indicated for cells that are contained in the cone of any erroneous output cell.
- When replacing references with constants, we do not immediately compute the concrete constant. Instead, we use the information, that there exists a constant that could eliminate the observed misbehavior. Only if we present a mutant to the user, we compute a concrete value for that constant. The reason for this delayed computation is the fact that there often exist many constants that satisfy the primary test case. During the distinguishing test case creation process, we gain additional information, which helps to reduce the number of constants.
- When changing references or resizing areas, we make use of the following assumption: If the user made a mistake when indicating the reference or area, the intended reference(s) might be in the surrounding of the originally indicated reference(s). We define the surrounding of a cell c as follows:

$$\textsc{Surround}(c) \equiv_{def} \left\{ c_1 \in CELLS \,\middle|\, \begin{array}{l} \varphi_x(c) - 2 \leq \varphi_x(c_1) \leq \varphi_x(c) + 2 \wedge \\ \varphi_y(c) - 2 \leq \varphi_y(c_1) \leq \varphi_y(c) + 2 \end{array} \right\}.$$

  We model into our CSP that the reference to the cell is either correct or that it should be replaced by one of the cells in the surrounding. In case of an area, we define the surrounding of the area as follows:

$$\textsc{Surround}(c_1 : c_2) \equiv_{def} \left\{ c_3 \in CELLS \,\middle|\, \begin{array}{l} \varphi_x(c_1) - 2 \leq \varphi_x(c_3) \leq \varphi_x(c_2) + 2 \wedge \\ \varphi_y(c_1) - 2 \leq \varphi_y(c_3) \leq \varphi_y(c_2) + 2 \end{array} \right\}.$$

  For areas, we allow to select/deselect any cell in the surrounding. This allows both, the shrinking and enlargement of areas and non-continuous areas.
- We allow only one mutation per cell.

These restrictions do not allow to find suited mutants for all given faulty spreadsheets. However, they allow the approach to be used in practice.

Example 4. The extended constraint representation for the cell $B6$ of our Cardiogenic shock estimator from Fig. 1 changes from $ab_{B6} \vee B6 = B2/B3$ to: $(ab_{B6} \wedge (B6 = B2{+}B3 \vee B6 = B2{-}B3 \vee B6 = B1/B3 \vee B6 = 5/B3 \vee \ldots)) \vee B6 = B2/B3$.

5 Computing distinguishing test cases

Usually, there exists more than one possible correction. In practice, a large number of repair suggestions overwhelms the user. Consequently, there is a strong need for distinguishing such explanations. One way to distinguish explanations is to use distinguishing test cases. Nica *et al.* [17] define a distinguishing test case for two variants of a program as input values that lead to the computation

of different output values for the two variants. When translating this definition to the spreadsheet domain, we have to search for constants that are assigned to inputs, which lead to different output values for the different explanations. The user (or another oracle) has to clarify which output values are correct.

Example 5. The following new input values form a distinguishing test case for the variants Π_1 and Π_2 of our running example: $\ell(\texttt{B2}) = \texttt{30}$, $\ell(\texttt{B3}) = \texttt{30}$, $\ell(\texttt{B4}) = \texttt{30}$, $\ell(\texttt{B5}) = \texttt{1}$. For these input values, Π_1 computes a value 0 for cell B8, where Π_2 would return 900.

Algorithm MUSSCO (Fig. 3) describes our overall approach. The algorithm takes a faulty spreadsheet and a failing test case as input and determines possible solutions with increasing cardinality. Since input cells are considered correct, the upper bound of the solutionSize is equal to the amount of non-input cells. In Line 1, the set TS initialized with the given failing test case. The sets eqM and udM are used to store the pairs of equivalent and undecidable mutants. The faulty spreadsheet and the given test cases are converted into constraints in Line 4. The function CONVERT slightly differs from the function described in Fig. 2: instead of only converting an expression into its constraint representation, also possible mutations are encapsulated in the constraint representation. The function GETSIZECONSTRAINT(Cons, n) creates a constraint that ensures at most n of the abnormal variables contained in Cons can be set to true (Line 5). In Line 6, the function HASSOLUTION checks if the solver can compute any mutants that satisfy the given constraint system. In Line 7, the function GETMUTANT returns a mutant that satisfies the given constraint system. This mutant is added to the list of mutants M (Line 8) and is blocked in the constraint system (Line 9). If M contains at least two mutants that are not equivalent or undecidable (Line 11), we call the test case retrieval function GETDISTTESTCASE with these mutants as parameters (Line 12). If this function returns UNSAT, the pair m_1, m_2 is added to the set eqMut (Line 14). If the function returns UNKNOWN, the pair m_1, m_2 is added to the set undesMut (Line 17). Otherwise, the function returns a new test case. The function GETEXPECTEDOUTPUT is used to determine the expected output for the given test case (Line 19). This function asks the user (or another oracle) for the expected output. The test case is added to the set of test cases (Line 20) and to the constraint system (Line 21). The function FILTER returns the set of mutants that fail this test case (Line 22). Those mutants are removed from the set of mutants (Line 23). After retrieving all mutants for the given solutionSize, the remaining solutions M are presented to the user. If the user accepts one solution, the algorithm terminates. Otherwise, the solutionSize is incremented (Line 31).

Algorithm GETDISTTESTCASE (Fig. 4) creates distinguishing test cases. This algorithm takes as input a spreadsheet and two mutated versions of that spreadsheet. The functions GETINPUTCELLS and GETOUTPUTCELLS return the set of input and output cells for a given spreadsheet (Lines 1 and 2). In Lines 3 and 4, the mutants m_1 and m_2 are converted into their constraint representations. When creating a distinguishing test case, we have to exclude the input

Input: A spreadsheet Π, a test case T
Output: A set of possible corrections

solutionSize = 1; TS = $\{T\}$
while solutionSize $\leq$ ($|\Pi|$ − |GetInputCells(Π)|) **do**
  $M = \{\}$; eqM ={}; udM ={}
  Cons =Convert(Π, TS)
  Cons = Cons ∪ GetSizeConstr(Cons, solutionSize)
  while HasSolution(Cons) **do**
    m = GetMutant(Cons)
    $M = M \cup \{m\}$
    Cons = Cons ∪{¬m}
    while $|M| \geq 2 \wedge \exists((m_1, m_2) \in M : (m_1, m_2) \notin \text{eqM} \wedge (m_1, m_2) \notin \text{udM})$ **do**
      Select m_1, m_2 from M where $(m_1, m_2) \notin \text{eqM} \wedge (m_1, m_2) \notin \text{udM}$
      T' = GetDistTestCase(Π, m_1, m_2)
      if T' = UNSAT **then**
        eqM = eqM ∪ $\{(m_1, m_2)\}$
      else
        if T' = UNKNOWN **then**
          udM = udM ∪ $\{(m_1, m_2)\}$
        else
          $T' = T' \cup$ GetExpectedOutput(Π, T')
          TS = TS ∪ $\{T'\}$
          Cons = Cons ∪ Convert(T')
          M' = Filter(Π, T', M)
          $M = M \setminus M'$
        end if
      end if
    end while
  end while
  if User accepts any solution in M **then**
    return M
  end if
  solutionSize = solutionSize + 1
end while
return no solution

Fig. 3: Algorithm Mussco(Π, T)

Input: A spreadsheet Π, mutants m_1, m_2
Output: A distinguishing test case or UNSAT/UNKOWN

inputCells = GetInputCells(Π)
outputCells = GetOutputCells(Π)
Cons1 =Convert($\Pi \setminus$ inputCells, m_1, "_1")
Cons2 =Convert($\Pi \setminus$ inputCells, m_2, "_2")
inputCon = $\bigwedge_{c \in inputCells} c\_1 = c\_2$
outputCon = $\bigvee_{c \in outputCells} c\_1 \neq c\_2$
Cons = Cons1 ∪ Cons2 ∪ inputCon ∪ outputCon
return GetSolution(Cons)

Fig. 4: Algorithm GetDistTestCase(Π, m_1, m_2)

cells from the spreadsheet. Therefore, we only hand over the spreadsheet without the input cells to the function CONVERT. This function slightly differs from the CONVERT function from Fig. 2, because it takes two additional parameters: (1) the particular mutant in use and (2) a constant that acts as postfix for variables. This postfix is necessary to distinguish the constraint representation of m_1 from that of m_2: Each variable in the constraint system for mutant m_1 (m_2) gets the postfix "_1" ("_2"). In Line 5, a constraint is created that ensures that the input of m_1 is equal to the input of m_2. In Line 6, a constraint is created that ensures that at least one output cell of m_1 has a different value than the same output cell in m_2. The function GETSOLUTION calls the solver with these constraints (Line 8). This function either returns a distinguishing test case, UNSAT (in case of equivalent mutants) or UNKOWN (in case of undecidability).

The worst-case time complexity of the Algorithm from Fig. 3 is exponential in the number of cells ($O\left(2^{|CELLS|}\right)$). In practice, only solutions up to a certain size, i.e. single or double fault solutions, are relevant. The algorithm terminates: The outer while-loop (Line 2) is bound to the size of the spreadsheet. The while-loop in Line 6 is limited since there only exists a limited number of mutants that can be created and we do not allow to report mutants twice (Line 9). The inner-most loop (Line 10) is limited since the number of mutants in M has to be greater or equal to two and the selected pair has not already been proven to be equivalent or undecidable. In each iteration of this loop, either a new pair is added to the equivalent or undecidable set (Lines 14 and 17) or the set M shrinks (Line 23). M must shrink because the return set of the function FILTER (Line 22) contains at least on element, since the mutants m_1 and m_2 must compute different output values for the given test case.

6 Empirical Evaluation

We implemented a prototype in Java that uses Z3 [7] as solver. This prototype supports the conversion of spreadsheets with basic functionality (arithmetic and relational operators, the functions 'IF', 'SUM', 'AVERAGE', 'MIN', and 'MAX') into Z3 formula clauses.

For the evaluation, we used the publicly available Integer Spreadsheet Corpus [5]. This corpus comes with 33 different spreadsheets (12 artificially created spreadsheets and 21 real-life spreadsheets) and 229 mutants of these 33 basic spreadsheets. We excluded some spreadsheets from our evaluation, because MUSSCO was not able to generate the required mutation to correct the observed misbehavior. There are two reasons for this: (1) The correction requires more than one mutation within a single cell, which is currently not supported by our approach. (2) The required mutation operator is not implemented in MUSSCO. In the following empirical evaluation, we only consider the 73 spreadsheets where MUSSCO was able to compute the required mutation in order to correct the fault.

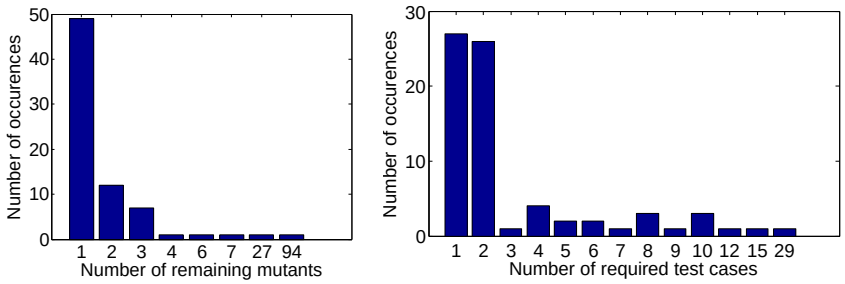
The original spreadsheets used in this empirical evaluation are listed in Table 1. Because of space limitations, we only list the original version of the spreadsheets, instead of each faulty version. This table indicates for each spreadsheet

Table 1: Structure and complexity of the evaluated spreadsheets

Name	**Number of cells**			**Halstead complexity**				
	In	**Out**	**Form.**	η_1	η_2	N_1	N_2	*difficulty*
amortization	16	1	16	5	33	31	67	5.4
arithmetics00	10	1	8	1	23	11	29	1.3
arithmetics01	9	1	11	2	23	14	34	2.4
arithmetics02	13	1	16	1	36	21	50	1.2
arithmetics03	19	1	35	1	64	45	99	1.1
arithmetics04	23	2	24	1	59	51	98	1.0
austrian_league	91	10	32	3	103	96	267	4.2
bank_account	45	13	27	7	76	103	187	6.4
birthdays	39	3	39	7	86	78	189	8.5
cake	101	1	69	3	155	69	238	5.2
comp_shopping	37	4	36	6	64	151	288	5.7
conditionals01	9	1	11	5	25	34	65	4.8
dice_rolling	31	3	21	4	40	99	190	3.8
fibonacci	25	1	46	1	68	16	87	2.7
matrix	51	1	13	3	23	17	67	5.9
oscars2012	60	2	22	3	76	24	104	6.5
prom_calculator	46	1	14	2	63	14	73	5.2
shares	43	12	39	4	69	37	118	6.4
shop_bedroom1	67	2	32	2	78	32	129	4.0
shop_bedroom2	70	4	64	4	109	148	338	4.6
training	34	3	53	4	93	99	223	4.5
weather	70	5	41	6	131	89	231	7.8
wimbledon2012	90	1	49	4	135	280	538	3.8
Average	**43.4**	**3.2**	**31.2**	**3.4**	**71.0**	**67.8**	**161.3**	**4.5**

the amount of input, output and formula cells. The smallest spreadsheet contains 8 formulas and the largest contains 69 formula cells. On average, a spreadsheet contains 31.2 formula cells. To express the complexity of the spreadsheets, we adapt the Halstead complexity measures [9] to the spreadsheet domain. η_1 represents the number of distinct operators that are used within a spreadsheet. η_2 is the number of distinct operands (i.e. cell references, constants) that are used within a spreadsheet. N_1 indicates the total number of operators while N_2 indicates the total number of operants. From these basic metrics, we derive the vocabulary ($\eta = \eta_1 + \eta_2$) and the spreadsheet length ($N = N_1 + N_2$). The average vocabulary is 74.4 and the average spreadsheet length is 229.1. An interesting Halstead metric is the difficulty ($D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$). The difficulty measure can be seen as the difficulty to understand the spreadsheet when debugging the spreadsheet. The difficulty of the investigated spreadsheets ranges from 1.0 to 8.5. The average difficulty is 4.5.

The faulty spreadsheet variants have on average 1.14 erroneous output cells. 52 mutated spreadsheets contain single faults. 20 mutated spreadsheets contain double faults, i.e. two cells with wrong formulas. One mutated spreadsheets contains three faults. The evaluation was performed using a PC with an Intel



(a) Remaining fixes (b) Generated dist. test cases

Fig. 5: Empirical results

Core i7-3770K CPU and 16GB RAM. The evaluation machine runs a 64-bit Windows 7 and the Oracle Java Virtual Machine version 1.7.0_17. We set a time limit of 2000 seconds (i.e. approximately 33 minutes) per faulty spreadsheet for generating mutants and distinguishing test cases. The evaluation results are averaged over 100 runs.

In order to investigate a larger amount of spreadsheets, we decided to simulate the user interactions. Therefore, we use the original correct spreadsheets as oracles to determine the output values for the generated distinguishing test cases.

Fig. 5a shows the amount of correction suggestions that are returned to the user. For 49 spreadsheets, only the correct mutation is returned to the user. On average, 3.2 mutants are reported to the user. For one faulty spreadsheet containing two faulty cells, MUSSCO determines 27 correction suggestions. Moreover, applying the algorithm to a spreadsheet with three faults results in 94 correction suggestions. The evaluation shows that in case of double or triple faults, MUSSCO finds a higher amount of equivalent solutions.

Fig. 5b illustrates the number of generated distinguishing test cases. For 27 spreadsheets, only a single distinguishing test case is required. For 26 spreadsheets, two distinguishing test cases are necessary. For one spreadsheet, 29 distinguishing test cases have to be generated. This spreadsheet contains a double fault. Therefore, MUSSCO creates many mutants which have to be killed by the distinguishing test cases. On average, 3.1 distinguishing test cases are required.

The average runtime is 49.1 seconds, at which the runtime is less than 10 seconds for 23 of the spreadsheets. The average runtime for single faults is 25.1 seconds. The average runtime for double and triple faults is 108.6 seconds. Most of the runtime, i.e. 95.5 % is consumed by the mutation creation process. The creation of the distinguishing test cases requires on average 1.4 % of the total run time. The remaining 3.1 % encompasses between the time required for filtering the mutants and setting up MUSSCO (read spreadsheet data in, convert spreadsheet).

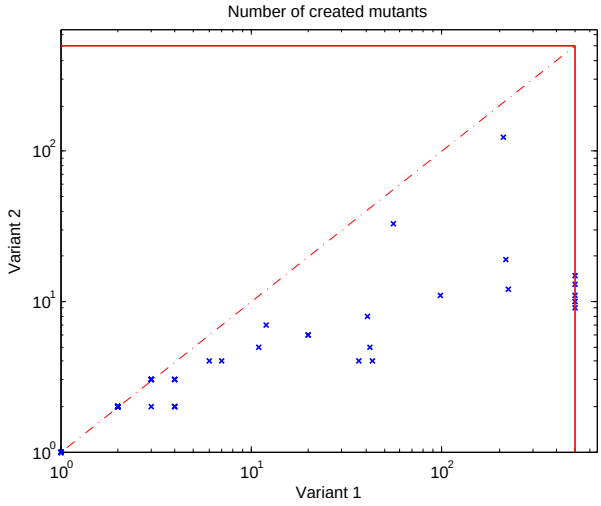


Fig. 6: Comparison of two computation variants w.r.t. the number of created mutants. Data points along the dashed line indicate that the variants generate the same number of mutants. Data points below the dashed line indicate that Variant 1 creates more mutants. The solid lines indicate the timeout.

We create a distinguishing test case as soon as we have two mutants available. Another possibility is to immediately compute all possible mutants of a particular size and afterwards generate the test cases. Does the implemented method perform better with respect to runtime? We suppose that adding more test cases to the constraint system decreases the number of mutants that are created and therefore decreases the total computation time. For clarifying our assumptions, we compare the two methods with respect to the number of generated mutants and the total computation time in the Fig. 6 and 7. Variant 1 denotes the version where we first compute all possible mutants. Variant 2 denotes the version described in Algorithm MUSSCO (Fig. 3). For six spreadsheets, Variant 1 results in a timeout. On average, Variant 1 creates 17.2 mutants while Variant 2 creates 5.2 mutants (when comparing only those spreadsheets without timeouts). However, when comparing the computation time (see Fig. 7), the two variants only slightly differ (expect for the six spreadsheets yielding a timeout when using Variant 1). It turns out, that decreasing the number of computed mutants through more test cases, increases the computation time per mutant. Nevertheless, we favor Variant 2 over Variant 1 since the user gets earlier a first response.

7 Related Work

Our approach is based on model-based diagnosis [20], namely its application to (semi-) automatic debugging. It uses a constraint representation and a constraint

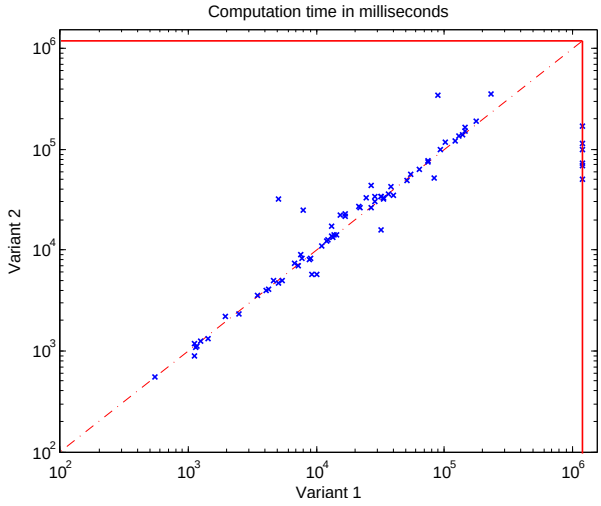


Fig. 7: Comparison of two computation variants w.r.t. the total computation time. Data points along the dash-dot line indicate that the variants perform equal w.r.t. runtime. Data points below the dashed line indicate that Variant 2 requires less computation time than Variant 1. The solid lines indicate the timeout.

solver. , which pinpoints software failures. Jannach and Engler [13] presented a model-based approach that uses an extended hitting-set algorithm and user-specified or historical test cases and assertions, to calculate possible error causes in spreadsheets.

GoalDebug [1] is a spreadsheet debugger for end-users. This approach generates a list of change suggestions for formulas that would result in a user-specified output. GoalDebug relies upon a set of pre-defined change inference rules. Hofer and Wotawa [12] also proposed an approach for generating repair candidates via genetic programming. In contrast to these approaches, we encode the mutation creation into a constraint satisfaction problem. In addition, we generate distinguishing test cases to keep the number of possible fixes small.

Ruthruff *et al.* [22] and Hofer *et al.* [10] propose to use spectrum-based fault localization for spreadsheet debugging. In contrast to MUSSCO, these approaches only identify the locations of faults instead of giving repair suggestions.

Spreadsheet testing is closely related to debugging. In the WYSIWYT system, users indicate correct/incorrect output values by placing a correct/faulty token in the cell [21]. The spreadsheet analysis tools of Abraham and Ewig [2] and Ahmad *et al.* [4] reason about the units of cells to find inconsistencies in formulas. The tools differ in the rules they employ and in the degree to which they require users to provide additional input. Ahmad's tool requires users to annotate the spreadsheet cells with additional information. UCheck [2] fully automatically performs unit analysis by exploiting techniques for automated header inference.

8 Conclusions

Our spreadsheet debugging approach Mussco maps a spreadsheet into a set of constraints for computing potential diagnosis candidates. The approach makes use of mutations, i.e., small changes of formulas used in the spreadsheets, to create diagnosis candidates. These diagnosis candidates are further refined by generating distinguishing test cases.

Beside the theoretical foundations and the algorithms we also discuss the results obtained from an empirical evaluation where we are able to show that distinguishing test cases improve diagnosis of spreadsheets substantially. In particular, results show that on average 3.1 distinguishing test cases are generated and 3.2 mutants are reported as possible fixes. On average, the generation of the mutants and distinguishing test cases requires 47.9 seconds in total, rendering the approach applicable as a real-time application. In future work, we will extend the toolset (i) by supporting more functionality of spreadsheets, and (ii) by integrating it into a spreadsheet framework.

Acknowledgement

The work described in this paper has been been funded by the Austrian Science Fund (FWF) project DEbugging Of Spreadsheet programs (DEOS) under contract number I2144 and the Deutsche Forschungsgemeinschaft (DFG) under contract number JA 2095/4-1.

References

1. Abraham, R., Erwig, M.: GoalDebug: A spreadsheet debugger for end users. In: Proceedings of the 29th International Conference on Software Engineering. ICSE '07, Washington, DC, USA, IEEE Computer Society (2007) 251–260
2. Abraham, R., Erwig, M.: UCheck: A spreadsheet type checker for end users. Journal of Visual Languages and Computing **18** (February 2007) 71–95
3. Abreu, R., Hofer, B., Perez, A., Wotawa, F.: Using constraints to diagnose faulty spreadsheets. Software Quality Journal **23**(2) (2015) 297–322
4. Ahmad, Y., Antoniu, T., Goldwater, S., Krishnamurthi, S.: A type system for statically detecting spreadsheet errors. In: 18th IEEE Int. Conference on Automated Software Engineering (ASE 2003), IEEE Computer Society (2003) 174–183
5. Ausserlechner, S., Fruhmann, S., Wieser, W., Hofer, B., Spork, R., Mühlbacher, C., Wotawa, F.: The right choice matters! SMT solving substantially improves model-based debugging of spreadsheets. In: 2013 13th International Conference on Quality Software (QSIC'13). (2013) 139–148
6. Chadwick, D., Knight, B., Rajalingham, K.: Quality control in spreadsheets: A visual approach using color codings to reduce errors in formulae. Software Quality Control **9**(2) (June 2001) 133–143
7. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 337–340

8. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
9. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY, USA (1977)
10. Hofer, B., Perez, A., Abreu, R., Wotawa, F.: On the empirical evaluation of similarity coefficients for spreadsheets fault localization. Journal of Automated Software Engineering - Special Issue on Realizing Artificial Intelligence and Software Engineering Synergies **22**(1) (2015) 47–74
11. Hofer, B., Riboira, A., Wotawa, F., Abreu, R., Getzner, E.: On the empirical evaluation of fault localization techniques for spreadsheets. In: 16th Int. Conference on Fundamental Approaches to Software Engineering (FASE 2013). Lecture Notes in Computer Science, Springer (2013) 68–82
12. Hofer, B., Wotawa, F.: Mutation-based spreadsheet debugging. In: International workshop on program debugging (IWPD), - Supplemental Proceedings ISSRE 2013. (2013) 132–137
13. Jannach, D., Engler, U.: Toward model-based debugging of spreadsheet programs. In: Proceedings of the 9th Joint Conference on Knowledge-Based Software Engineering. JCKBSE'10, Kaunas, Lithuania (2010) 252–264
14. Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S.: The state of the art in end-user software engineering. ACM Comput. Surv. **43**(3) (April 2011) 21:1–21:44
15. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning (JAR) **40**(1) (January 2008) 1–33
16. Liffiton, M.H., Sakallah, K.A.: Generalizing core-guided max-sat. In: Proceedings of the 12th Int. Conference on Theory and Applications of Satisfiability Testing. SAT '09, Berlin, Heidelberg, Springer-Verlag (2009) 481–494
17. Nica, M., Nica, S., Wotawa, F.: On the use of mutations and testing for debugging. Software : practice & experience (2012) http://dx.doi.org/10.1002/spe.1142.
18. Panko, R.R.: Applying code inspection to spreadsheet testing. Journal of Management Information Systems **16**(2) (1999) 159–176
19. Panko, R.R., Port, D.: End user computing: The dark matter (and dark energy) of corporate IT. In: 45th Hawaii International International Conference on Systems Science (HICSS-45 2012), Proceedings. (2012) 4603–4612
20. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1) (1987) 57–95
21. Rothermel, K.J., Cook, C.R., Burnett, M.M., Schonfeld, J., Green, T.R.G., Rothermel, G.: WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation. In: Proceedings of the 22nd International Conference on Software engineering. ICSE '00, New York, NY, USA, ACM (2000) 230–239
22. Ruthruff, J., Creswick, E., Burnett, M., Cook, C., Prabhakararao, S., Fisher, II, M., Main, M.: End-user software visualizations for fault localization. In: Proceedings of the 2003 ACM Symposium on Software visualization. SoftVis '03, New York, NY, USA, ACM (2003) 123–132
23. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering. ICSE '09, Washington, DC, USA, IEEE Computer Society (2009) 364–374