Noname manuscript No.
(will be inserted by the editor)

Using Constraints to Diagnose Faulty Spreadsheets

Rui Abreu · Birgit Hofer · Alexandre Perez · Franz Wotawa

the date of receipt and acceptance should be inserted later

Abstract Spreadsheets can be viewed as a highly flexible programming environment for end-users. Spreadsheets are widely adopted for decision making, and may have a serious economical impact for the business. However, spreadsheets are staggeringly prone to errors. Hence, approaches for aiding the process of pinpointing the faulty cells in a spreadsheet are of great value. We present a constraint-based approach, ConBug, for debugging spreadsheets. The approach takes as input a (faulty) spreadsheet and a test case that reveals the fault and computes a set of diagnosis candidates for the debugging problem. Therefore, we convert the spreadsheet and a test case to a constraint satisfaction problem. We perform an empirical evaluation with 78 spreadsheets from different sources, where we demonstrate that our approach is light-weight and efficient. From our experimental results, we conclude that ConBug helps end-users to pinpoint faulty cells.

Keywords Spreadsheets · Debugging · Constraints

1 Introduction

Spreadsheet tools, such as Microsoft Excel[1], iWork's Numbers[2], and OpenOffice's Calc[3], can be viewed as programming environments for non-professional program-

Rui Abreu · Alexandre Perez
Dept. of Informatics Engineering, University of Porto, Porto, Portugal
E-mail: rui@computer.org, alexandre.perez@fe.up.pt

Birgit Hofer · Franz Wotawa
Institute for Software Technology, Graz University of Technology, Graz, Austria
E-mail: bhofer@ist.tugraz.at, wotawa@ist.tugraz.at

[1] http://office.microsoft.com/en-gb/excel/
[2] http://www.apple.com/iwork/numbers/
[3] http://www.openoffice.org/product/calc.html

mers [27]. These so-called "end-user" programmers[4] vastly outnumber professional ones: the US Bureau of Labor and Statistics estimates that more than 55 million people use spreadsheets and databases at work on a daily basis [27,22,23]. Despite this trend, as a programming language, spreadsheets lack support for abstraction, testing, encapsulation, or structured programming. As a consequence, spreadsheets are error-prone. As a matter of fact, numerous studies have shown that existing spreadsheets contain redundancy and errors at an alarmingly high rate [14,38]. As an example disastrous financial consequences due to spreadsheet calculating errors, the Board of the West Baraboo Village, USA, found out on December 9, 2011 that they will be paying $400,0000 more on the estimated total cost for the 10-year borrowing than originally projected[5].

In the software engineering domain, constraints have been used for various purposes like verification [15], debugging [13,41], program understanding [40] as well as testing [20,21]. Some of the proposed techniques use constraints to state specification knowledge like pre- and post-conditions. Others use constraints for modeling purposes or extract the constraints directly from the source code.

In this paper, we propose a constraint-based approach for debugging spreadsheets, dubbed ConBug. This paper is an extension of the work published in [7], [8] and [25]. The approach takes as input a spreadsheet and the set of user expectations, and produces as output a set of diagnosis candidates. User expectations express the cells that, according the user, reveal failures on the spreadsheet. Diagnosis candidates are explanations for the misbehavior in user expectations. The main contribution of this paper is the profound explanation of the approach and an extended empirical evaluation using spreadsheets from different sources. This paper extends previous work [7], [8], [25] as follows:

- The description of the ConBug approach has been expanded. In contrast to previous work, we now explain the conversion different types of expressions.
- All algorithms are enhanced by a time complexity and termination analysis.
- The running example to illustrate basic concepts has been improved.
- The modeling of a spreadsheet constraint satisfaction problem with the constraint solver Minion is discussed in detail.
- The empirical evaluation of our approach is extended from 5 to 78 spreadsheets.
- A comparison of ConBug to spectrum-based fault localization is given in the empirical evaluation.

In the remainder of this paper, we make use of a running example to illustrate the basic concepts of ConBug. Despite of being a contrived example, it serves well the purpose of demonstrating how our approach works. Figure 1 shows this example spreadsheet stemming from the EUSES spreadsheet corpus [18]. This spreadsheet is used to calculate the wages of the workers (cells `F2:F3`) and the total working hours (cell `D4`). For the sake of clarity, we have reduced the number of columns and rows compared to the original spreadsheet. In this example

[4] We are aware that there exist different types of spreadsheet usage scenarios: quickly written calculations and spreadsheets carefully written for long-term use. The first type often contains careless mistakes that can be easily detected with the help of pattern identification mechanisms that are already implemented in most of the available spreadsheet tools. The second spreadsheet type often contains faults that are difficult to detect and localize. In this paper, we therefore focus on the localization of the latter one.

[5] http://www.wiscnews.com/baraboonewsrepublic/news/local/article_7672b6c6-22d5-11e1-8398-001871e3ce6c.html

	A	B	C	D	E	F
1		week 1	week 2	Total	$/h	Gross Pay
2	Green	23	31	54	15	$810
3	Jones	35	34	69	17	$1 173
4	Total	58	65	123		
5						

(a) Correct spreadsheet

	A	B	C	D	E	F
1		week 1	week 2	Total	$/h	Gross Pay
2	Green	23	31	23	15	$345
3	Jones	35	34	69	17	$1 173
4	Total	58	65	92		
5						

(b) Faulty spreadsheet

	A	B	C	D	E	F	G
1		week 1	week 2	Total	$/h	Gross Pay	
2	Green	23	31	=SUM(B2)			Should be B2:C2
3	Jones	35	34	=SUM(B3+C3)	17	=D3*E3	
4	Total	=SUM(B2:B3)	=SUM(C2:C3)	=SUM(D2:D3)			
5							

(c) Formula view of Figure 1(b)

Fig. 1 Running example stemming from the EUSES spreadsheet corpus [18]

spreadsheet, users may quickly pinpoint the source of the problems by manually inspecting the formulas, but this spreadsheet serves well to illustrate how our approach works. Figure 1(a) illustrates the correct version of this spreadsheet, Figure 1(b) a faulty variant of the same spreadsheet.

Figure 1(c) shows the formula view of the faulty spreadsheet from Figure 1(b). In this faulty spreadsheet, the computation of the total hours for the worker "Green" (cell `D2`) is faulty because the programmer of the spreadsheet unintentionally set a wrong area for the SUM formula. This happens for example when a programmer adds a new week but forgets to adapt some calculations. Because of this fault, the wage of the worker "Green" (cell `F2`) and the total hours (cell `D4`) compute the wrong values (observed failures). An example of a diagnosis candidate for this concrete example is cell `D2` or the cells `D4` and `F2` together.

The remainder of this paper is organized as follows: Section 2 deals with the basic definitions required in the spreadsheet domain. The spreadsheet debugging problem and the constraint satisfaction problem are stated. Section 3 explains the conversion of a spreadsheet debugging problem into a constraint satisfaction problem. In Section 4, we introduce an algorithm for computing diagnosis candidates given a constraint satisfaction debugging problem. The design and the results of the empirical evaluation are discussed in Section 5. The evaluation comprises single, double and triple faults. In addition, the conversion of spreadsheets into MINION constraints is demonstrated. Section 6 deals with the related work and Section 7 concludes the paper and discusses future work.

2 Basic Definitions

The aim of this section is to define the spreadsheet debugging problem. Therefore, we assume a spreadsheet programming language $\mathcal{L}$ with syntax and semantics similar to, e.g., Microsoft Excel. In order to be self-contained, we adopt a subset of the definitions of the syntax and semantics for $\mathcal{L}$ from [25] to our approach.

A spreadsheet consists of cells. Each cell c has an expression $\ell(c)$ and a value $\nu(c)$. The expression of a cell $\ell(c)$ can either be empty or an expression written in the language $\mathcal{L}$. If no expression is explicitly declared for a cell, the

function ℓ returns the value 0. The value of a cell c is determined by its expression. It can be either undefined ϵ, an error $\perp$, or any number, boolean or string value.

Each cell can be accessed by its column and row number. In most spreadsheet languages, the rows are numbered whereas columns have a corresponding letter. For example, `A3` denotes the cell at column 1 and row 3. For simplicity, we assume a function φ that maps the cell names from a set *CELLS* to their corresponding position (x, y) in the matrix where x represents the column and y the row number. The functions φ_x and φ_y return the column and row number of a cell respectively.

For referencing several cells at once, areas are used. We define areas as follows:

Definition 1 An area is a set consisting of all cells that are within the area that is spanned by the cells $c_1, c_2 \in CELLS$. Formally:

$$c_1{:}c_2 \equiv_{def} \left\{ c \in CELLS \,\middle|\, \begin{array}{l} \varphi_x(c_1) \leq \varphi_x(c) \leq \varphi_x(c_2) \;\; \& \\ \varphi_y(c_1) \leq \varphi_y(c) \leq \varphi_y(c_2) \end{array} \right\} \tag{1}$$

Obviously, every area is a subset of the set of cells ($c_1{:}c_2 \subseteq CELLS$).

Below, we introduce the language $\mathcal{L}$ for representing expressions that are used to compute values for cells. This language takes the values of cells and constants together with operators and conditionals to compute values for other cells. The language is a functional language, i.e., only one value is computed for a specific cell. Moreover, recursive functions are not allowed.

Definition 2 (Syntax of $\mathcal{L}$) The syntax of $\mathcal{L}$ is recursively defined as follows:

- Constants k representing ϵ, number, boolean, or string values are elements of $\mathcal{L}$ (i.e., $k \in \mathcal{L}$).
- All cell names are elements of $\mathcal{L}$ (i.e., $CELLS \subset \mathcal{L}$).
- Areas $c_1 : c_2$ are elements of $\mathcal{L}$.
- If $e_1, e_2, \ldots, e_n$ are elements of the language ($e_1, e_2, \ldots, e_n \in \mathcal{L}$), then the following expressions are also elements of $\mathcal{L}$:
  - (e_1) is an element of $\mathcal{L}$.
  - If o is an operator ($o \in \{+, \text{-}, *, /, <, =, >\}$), then $e_1\ o\ e_2$ is an element of $\mathcal{L}$.
  - A function call $f(e_1, \ldots, e_n)$ is an element of $\mathcal{L}$ where f denotes functions like **IF**, **SUM**, **AVG**, etc.

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

Definition 3 (Semantics of $\mathcal{L}$) Let e be an expression from $\mathcal{L}$ and ν a function mapping cell names to values. We define the semantic of $\mathcal{L}$ recursively as follows:

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

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

– If e is of the form $f(e_1,\ldots,e_n)$, then the value returned by the implementation of the function f is returned. Let f_I be the implementation of the function f. The semantics of the call to a function is defined as follows:
$[\![f(e_1,\ldots,e_n)]\!] = f_I([\![e_1]\!],\ldots,[\![e_n]\!])$
The return value might be $\perp$ in case of type errors or mismatches of arguments.

The function $\rho : \mathcal{L} \mapsto 2^{\text{CELLS}}$ returns the set of referenced cells. Formally, we define ρ as follows:

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

– If e is a constant, then $\rho(e) = \emptyset$.
– If e is a cell c, then $\rho(e) = \{c\}$.
– If $e = (e_1)$, then $\rho(e) = \rho(e_1)$.
– If $e = e_1\ o\ e_2$, then $\rho(e) = \rho(e_1) \cup \rho(e_2)$.
– If $e = f(e_1,\ldots,e_n)$, then $\rho(e) = \bigcup_{i=1}^{n} \rho(e_i)$.

Definition 5 (The function CONE) Given a spreadsheet Π and a cell $c \in \Pi$, we define the function CONE recursively as follows:

$$\text{CONE}(c) = c \cup \bigcup_{c' \in \rho(c)} \text{CONE}(c') \tag{2}$$

Example 1 *The cone for cell D4 from our running example from Figure 1(c) is* $\text{CONE}(D4) = \{B2, B3, C3, D2, D3, D4\}$.

Having defined the fundamentals of spreadsheets, we are now able to focus on the aspects of debugging spreadsheets. For this, we define the terms direct data dependency, input and output as follows:

Definition 6 (Direct data dependency) A cell c is direct data dependent on another cell c' if and only if the cell c' is referenced in $\ell(c)$:

$$dd(c', c) \Leftrightarrow c' \in \rho(l(c))$$

Example 2 *For our running example from Figure 1(c), we have the following direct data dependencies (amongst other data dependencies):* $dd(B2, B4)$ *and* $dd(B3, B4)$.

Definition 7 (Input/Output cell) An input cell c is a cell that does not reference another cell c', i.e. there exists no direct data dependency from any cell c' to c. Conversely, an output cell c is a cell that is not referenced by another cell c', i.e. there exists no direct data dependency from c to any other cell c'.

$$Input(\Pi) = \{c|(\nexists dd(c', c) \wedge \exists cr : CELLS|dd(c, cr))\} \tag{3}$$

$$Output(\Pi) = \{c|\nexists dd(c, c')\} \tag{4}$$

Example 3 *According to the formula view from our running example (see Figure 1(c)), we have six input cells (*`B2:C3` *and* `E2:E3`*) and five output cells (*`B4:D4` *and* `F2:F3`*). For the sake of clarity, we do not consider cells containing strings in this example.*

For defining test cases, we introduce environments:

Definition 8 (Environment) An environment is a set of pairs (x, v) where x is a cell and v its value. In an environment there is only one pair for a cell.

With the definition of input and output cells and environments, we are able to define the terms test case and failing test case as follows:

Definition 9 (Test case) A test case for a spreadsheet $\Pi \in \mathcal{L}$ is a tuple (I, O) where I is the input environment specifying the values of all input cells used in Π, and O the output environment defining the expected output values (not necessarily specifying values for all output cells). An output cell for which no value is specified in O can have an arbitrary value.

Example 4 *For example, a test case for the spreadsheet from Figure 1 is*

$$I_\Pi : \{\texttt{B2} = 23;\ \texttt{C2} = 31;\ \texttt{B3} = 35;\ \texttt{C3} = 34;\ \texttt{E2} = 15;\ \texttt{E3} = 17\},\ \textit{and}$$

$$O_\Pi : \{\texttt{F2} = 810;\ \texttt{F3} = 1173;\ \texttt{B4} = 58;\ \texttt{C4} = 65;\ \texttt{D4} = 123\}.$$

Definition 10 (Failing test case) A test case is failing if there exists at least one output cell which calculated value differs from the expected value.

Example 5 *For the spreadsheet from Figure 1(c) the test case given in Example 4 is a failing test case because the computed output for the cells* `D4` *(*$D4 = 92$*) and* `E2` *(*$E2 = 345$*) differ from the expected output (*$D4 = 123;\ E2 = 810$*).*

From the definition of failing test cases, we derive the definition of passing test cases as follows:

$$\neg(\Pi \text{ fails test case}(I, O)) \Leftrightarrow \Pi \text{ passes test case } (I, O) \tag{5}$$

With these definitions, we are able to state the spreadsheet debugging problem:

Definition 11 (Spreadsheet debugging problem) Let $\Pi \in \mathcal{L}$ be a spreadsheet and T a failing test case of Π, then (Π, T) is a debugging problem.

A solution to the debugging problem is the identification of parts of the spreadsheet (set of cells) responsible for the detected misbehavior. We call such a set of cells an explanation. We have defined the debugging problem as a fault localization problem. Therefore, ConBug is designed as an approach to pinpoint certain cells of the spreadsheet. However, ConBug does not make suggestions how to change these cells so that the spreadsheet Π passes all test cases.

Many approaches are capable of returning explanations including [32,6,9,39, 28] and [13,42] among others. These approaches are designed for hardware description languages and imperative programming languages. Most of them focus on the fault localization process. Weimer et al. [39] propose an approach that delivers suggestions how to change a program in order to correct a fault. In this paper, we follow the debugging approach based on constraints, i.e., [13,42]. In particular, the approach makes use of the program's constraint representation to compute possible fault candidates. Therefore, debugging is reduced to solving the corresponding constraint satisfaction problem (CSP).

Definition 12 (Constraint Satisfaction Problem (CSP)) A constraint satisfaction problem is a tuple (V, D, CON) where V is a set of variables defined over a set of domains D connected to each other by a set of arithmetic and boolean relations, called constraints CON. A solution for a CSP represents a valid instantiation of the variables V with values from D such that none of the constraints from CON is violated.

The variables used in a CSP are not necessarily cells used in a spreadsheet. We discuss the representation of spreadsheets as a CSP in the next section.

3 CSP representation of spreadsheets

In order to solve the previously stated debugging problem, we have to convert spreadsheets into constraints. There exist some differences between the conversion of ordinary sequential programs and the conversion of spreadsheets: In [42,30], the authors introduce the conversion based on three steps: (1) removing loops, (2) providing a static single assignment form and (3) final compilation to constraints. In the domain of spreadsheets, there are (in almost all practical cases) no loops and every cell can only be defined once. Hence, there is no need for loop removal and the static single assignment form.

The basic idea of the conversion of the content of a spreadsheet into constraints is to use equations instead of assignments. The advantage of using equations instead of assignments is the directions of calculations: Assignments allow to deduce from the input to the output, but not vice versa. In contrast, equations allow to derive conclusions in both directions.

Example 6 *The cell* `F3` *from our running example from Figure 1(c) contains the expression* $\ell(F3) = D3 * E3$*. Instead of using an assignment form (*$F3 = D3 * E3$*), we use an equation:* $F3 == D3 * E3$*. This allows to deduce from the value of* `F3` *to the value of* `D3` *or* `E3`*.*

Algorithm ConvertExpression (Algorithm 1) illustrates the conversion. A constant or cell is represented by itself (Lines 1 to 3). Expressions in parentheses are converted by converting the expressions without the parentheses (Lines 4 to 7). For an expression of the form $e_1\ o\ e_2$, we convert e_1 and e_2 separately into constraints, and assign a new intermediate variable for each converted sub-expression (Lines 8 to 14). Therefore, an expression $e \in \mathcal{L}$ might be translated into several constraints. Representative for the functions f, we demonstrate the conversion of the functions **IF**, **SUM**, and **AND**:

- Let $\psi(cond, e_1, e_2, result)$ be a constraint that ensures the relationship of $cond, e_1, e_2$ and $result$ as follows: If $cond$ is true, $result$ must be equal to the value of e_1. Otherwise $result$ must be equal to the value of e_2 (Lines 15 to 21).
- Let $\text{SUM}(c_1...c_2, result)$ be a constraint that ensures that the sum of the values contained in the area c_1:c_2 is equal to the value of $result$ (Lines 22 to 25).
- Let $\text{AND}(c_1...c_2, result)$ be a constraint that ensures that $result$ = true if all c_n are true otherwise $result$ = false (Lines 26 to 29).

The other functions can be straightforwardly converted making use of appropriate equations and other constraints.

Algorithm 1 ConvertExpression

Require: Expression $e \in \mathcal{L}$
Ensure: [CON, *var*] with CON as a set of constraints, and *var* as the name of an auxiliary variable, a cell name or a constant
1: **if** e is a cell name or constant **then**
2: **return** [$\emptyset$, e]
3: **end if**
4: **if** e is of the form **(**e**)** **then**
5: Let [CON,aux] = ConvertExpression(e)
6: **return** [CON, aux]
7: **end if**
8: **if** e is of the form e_1 o e_2 **then**
9: Let [CON_1,aux_1] = ConvertExpression(e_1)
10: Let [CON_2,aux_2] = ConvertExpression(e_2)
11: Generate a new variable *result*
12: Create a new constraint CON accordingly to the given operator o, which defines the relationship between aux_1, aux_2, and *result*
13: **return** [$\text{CON}_1 \cup \text{CON}_2 \cup \text{CON}$, *result*]
14: **end if**
15: **if** e is of the form **if(**e_1**;**e_2**;**e_3**)** **then**
16: Let [CON_1,aux_1] = ConvertExpression(e_1)
17: Let [CON_2,aux_2] = ConvertExpression(e_2)
18: Let [CON_3,aux_3] = ConvertExpression(e_3)
19: Generate a new variable *result*
20: **return** [$\text{CON}_1 \cup \text{CON}_2 \cup \text{CON}_3 \cup \Psi(aux_1, aux_2, aux_3, result)$, *result*]
21: **end if**
22: **if** e is of the form **sum(**c_1**:**c_2**)** **then**
23: Generate a new variable *result*
24: **return** [SUM(c_1...c_2, *result*), *result*]
25: **end if**
26: **if** e is of the form **and(**c_1**:**c_2**)** or **and(**$c_1, \ldots, c_2$**)** **then**
27: Generate a new variable *result*
28: **return** [AND(c_1...c_2, *result*), *result*]
29: **end if**

For encoding of the debugging problem, we introduce a special boolean variable AB(c) for a cell c, that represents the 'health' status of the cell c: either the formula in c is correct or AB(c) = *true*. More formally:

$$\texttt{AB(c)} \vee \texttt{con}_\texttt{c}$$

assuming con_c is the constraint encoding of the content stored in cell c in the constraint programing language.

Algorithm ConvertSpreadsheet (Algorithm 2) illustrates the conversion of a Spreadsheet Π into a set of constraints. ConvertSpreadsheet takes a spreadsheet as input and returns a set of constraints as output. For all cells contained in Π the formulae are recursively converted into constraints using the function ConvertExpression (Algorithm 1). The returned variable of ConvertExpression is set equal to the current cell and connected to the abnormal variable AB(c) through a logical or. Finally, the algorithm returns all created constraints.

Algorithm 2 ConvertSpreadsheet

Require: Spreadsheet Π
Ensure: Set of constraints CON_Π representing the spreadsheet Π
1: Let CON_Π be an empty set
2: **for** cell $c \in \Pi$ **do**
3: **if** c is a formula cell **then**
4: $[con_1, aux]$ = ConvertExpression($\ell(c)$)
5: $con_c = \text{EQUAL}(c, aux)$
6: $\text{CON}_\Pi = \text{CON}_\Pi \cup con_1 \cup (AB_c \vee con_c)$
7: **end if**
8: **end for**
9: **return** CON_Π

Example 7 *When applying Algorithm 2 to our running example from Figure 1(c), we obtain the following constraints:*

$$\begin{aligned} CON_\Pi = \{\ & SUM(B2, B3, result_1) \\ & AB_{B4} \vee EQUAL(B4, result_1) \\ & SUM(C2, C3, result_2) \\ & AB_{C4} \vee EQUAL(C4, result_2) \\ & SUM(B2, result_3) \\ & AB_{D2} \vee EQUAL(D2, result_3) \\ & SUM(B3, C3, result_4) \\ & AB_{D3} \vee EQUAL(D3, result_4) \\ & SUM(D2, D3, result_5) \\ & AB_{D4} \vee EQUAL(D4, result_5) \\ & MULT(D2, E2, result_6) \\ & AB_{F2} \vee EQUAL(F2, result_6) \\ & MULT(D3, E3, result_7) \\ & AB_{F3} \vee EQUAL(F3, result_7)\}. \end{aligned}$$

The CSP representation of a program Π is given by the tuple $(V_\Pi, D, \text{CON}_\Pi)$. V_Π represents all non-empty cells of a program Π and all auxiliary variables created when calling the function ConvertExpression. The variables are defined over the domains $D = \{Integer, Boolean\}$[6].

Example 8 *The CSP representation of our running example from Figure 1(c) consists of CON_Π from Example 7 enhanced by the used variables V_Π and their domains D:*

$$\begin{aligned} V_\Pi = \{\ & AB_{B4}, AB_{C4}, AB_{D2}, AB_{D3}, AB_{D4}, AB_{F2}, AB_{F3} \\ & B2, B3, B4, C2, C3, C4, D2, D3, D4, E2, E3, F2, F3 \\ & result_1, result_2, result_3, result_4, result_5, result_6, result_7\} \\ D = \{\ & AB_{B4}, AB_{C4}, AB_{D2}, AB_{D3}, AB_{D4}, AB_{F2}, AB_{F3} : BOOLEAN \\ & B2, B3, B4, C2, C3, C4, D2, D3, D4, E2, E3, F2, F3 : INTEGER \\ & result_1, result_2, result_3, result_4, result_5, result_6, result_7 : INTEGER\}. \end{aligned}$$

The time complexity of Algorithm 2 is $O(|\Pi| * |e|)$ where $|\Pi|$ is the number of cells in the spreadsheet and $|e|$ the maximum size of the expressions used in

[6] In principle, other domains like Real numbers are possible. For the sake of clarity, we restrict the domains to Integer and Boolean in this paper. This is not a general limitation of the approach.

the cells. This expression size corresponds to the number of operators, constants, and references in the expression. The Algorithm once iterates over all cells (i.e., $|\Pi|$ cells) and invokes Algorithm 1 for each cell. The complexity of the conversion of an expression depends on the complexity of the expression itself. Therefore, the complexity of Algorithm 1 is $O(|e|)$. In theory, expressions can be arbitrary deeply nested and they can have an arbitrary number of subexpressions. However, in practice the nesting of formulas are limited. Moreover, for every spreadsheet there is a maximum expression size. Algorithms 1 and 2 clearly terminate since spreadsheets consist of a finite number of cells and expressions consist of a finite number of sub-expressions.

4 Debugging

Debugging of a spreadsheet requires the existence of a failing test case. Therefore, we need a set of constraints encoding a failing test case (I, O): For all $(x, v) \in I$ the constraint EQUAL(x, v) is added to the constraint system. For all $(y, w) \in O$ the constraint EQUAL(y, w) is added. Let CON_{TC} denote the constraints resulted from converting the given test case. Then, the CSP corresponding to the debugging problem of a spreadsheet Π is now represented by the tuple

$$(V_\Pi, D, \text{CON}_\Pi \cup \text{CON}_{TC}).$$

For convenience, we assume a function ConvertTest that implements the conversion of the failing test case into constraints: ConvertTest takes the failing test case as input and returns a set of constraints as output.

Example 9 *For our running example from Figure 1 and the test case from Example 4, the function* ConvertTest *returns the following constraints:*

$$\begin{aligned} CON_{TC} = \; & EQUAL(B2, 23) \\ & EQUAL(C2, 31) \\ & EQUAL(B3, 35) \\ & EQUAL(C3, 34) \\ & EQUAL(E2, 15) \\ & EQUAL(E3, 17) \\ & EQUAL(F2, 810) \\ & EQUAL(F3, 1173) \\ & EQUAL(B4, 58) \\ & EQUAL(C4, 65) \\ & EQUAL(D4, 123). \end{aligned}$$

Algorithm ConBug (Algorithm 3) illustrates the debugging process. This algorithm consists of three main phases: The first phase comprises the conversion of a spreadsheet $\Pi \in \mathcal{L}$ into the corresponding set of constraints (Line 6). Instead of converting the whole spreadsheet, only those cells are converted that are part of a cone of any cell indicated in the test case. The second phase is the conversion of the failing test case into the corresponding set of constraints (Line 7).

The third phase comprises the computation of diagnosis candidates, i.e., cells of the spreadsheet that might cause the revealed misbehavior (Lines 9 to 17).

Algorithm 3 Algorithm ConBug

Require: A spreadsheet Π and a failing test case T
Ensure: Diagnostic Report D
1: $D \leftarrow \emptyset$
2: cones $\leftarrow \emptyset$
3: **for all** cell $c \in T$ **do**
4: cones = cones $\cup$ Cone(c)
5: **end for**
6: $\text{CON}_{\Pi} \leftarrow$ ConvertSpreadsheet(cones)
7: $\text{CON}_{TC} \leftarrow$ ConvertTest(T)
8: $i \leftarrow 1$
9: **while** $i \leq |\Pi|$ **do**
10: $\text{CON}_{AB} = \{\text{COUNT}(\text{AB}, i)\}$
11: $D \leftarrow$ ConstraintSolver($\text{CON}_{\Pi} \cup \text{CON}_{TC} \cup \text{CON}_{AB}$)
12: **if** $D \neq \emptyset$ **then**
13: **return** D
14: **else**
15: $i \leftarrow i + 1$
16: **end if**
17: **end while**
18: **return** D

Therefore, the algorithm calls a constraint solver using the constraints CON_{Π}, CON_{TC} and CON_{AB}. The set of constraint CON_{AB} is responsible for allowing only a certain number of abnormal variables AB_c to be true. Therefore, we assume that the constraint solver offers a constraint $\text{COUNT}(\text{AB}, \mathit{result})$ that ensures that the number of variables of the type AB with true as value is equal to the value of *result*. The constraint solver returns all possible combinations of values of the abnormal variables AB_c so that the constraints CON_{Π}, CON_{TC} and CON_{AB} are not violated. The size (cardinality) of a solution corresponds to the size of the bug, i.e., the number of cells that must be changed in order to correct the fault. We assume that single cell bugs are more likely than bugs comprising more cells. Hence, we ask the constraint solver for smaller solutions first. If no solution of a particular size is found, the algorithm increases the size of the solutions to be searched for. This is done until either a solution is found or the maximum size of a bug, which is equivalent to the number of formula cells in Π, is reached.

Example 10 *Applying Algorithm 3 to the spreadsheet Π from Figure 1 and the test case from Example 4 works as follows: First, we compute the cones for the output variables (*$\text{Cone}(B4) = \{B2, B3, B4\}$, $\text{Cone}(C4) = \{C2, C3, C4\}$, $\text{Cone}(D4) = \{B2, B3, C3, D2, D3, D4\}$, $\text{Cone}(F2) = \{B2, D2, E2, F2\}$, $\text{Cone}(F3) = \{B3, C3, D3, E3, F3\}$*) and the set of all contained cells (cones =* $\{B2, B3, B4, C2, C3, C4, D2, D3, D4, E2, E3, E4, F2, F3\}$*). Afterwards, we create the set of constraints CON_{Π}, CON_{TC}, and CON_{AB} containing the constraint $\text{SUM}(AB_{B4}, AB_{C4}, AB_{D2}, AB_{D3}, AB_{D4}, AB_{F2}, AB_{F3}, 1)$[7]. The last mentioned constraint ensures that the solver returns single fault explanations first. Subsequently, the solver is called with these constraints. In case of our running example, the solver returns a single fault explanation, i.e. $D = \{AB_{D2}\}$. Therefore, the algorithm has found a solution and terminates.*

[7] Boolean values are interpreted as 1 if set to true. Otherwise they are interpreted as 0.

Algorithm 3 has a worst-case time complexity of $O(|\Pi| \cdot 2^{|\Pi| \cdot |e|})$. The computation of the cones (Lines 3 to 5) requires $O(\Pi^2)$ time as a cone requires computation of all referenced cells - which could be $|\Pi|$ cones in the worst case. The spreadsheet can be converted (Line 6) in $O(|\Pi| * |e|)$ time (as explained in Section 3). The test case can be converted (Line 7) in $O(|\Pi|)$ time since the number of cells in the test case must be less than $|\Pi|$. The creation of one set of constraints CON_{AB} (Line 10) requires $O(|\Pi|)$ since there exist at most $|\Pi|$ abnormal variables. However, it is well known that constraint solving is NP-complete. Therefore, this part of the algorithm is the one that has the greatest impact on the worst-case complexity. Calling a constraint solver with $O(|\Pi| \cdot |e|)$ variables (Line 11) requires $O(2^{|\Pi| \cdot |e|})$ time in order to find a satisfiable value assignment. Since the constraint solver call is in a loop, we finally obtain the worst-case time complexity of $O(|\Pi| \cdot 2^{|\Pi| \cdot |e|})$ for our algorithm. It is worth noting that in practice constraint solving is much faster and depends heavily of the structure of the constraint satisfaction problem. For example, if the constraint satisfaction problem is acyclic a solution can be provided in polynomial time. Moreover, in the debugging domain someone is mainly interested in finding single, double or triple faults at most. Hence, the loop from Line 9 to 17 can be neglected.

Algorithm 3 terminates if the spreadsheet does not contain any circular references[8]. Otherwise the computation of the cones (Line 3) could result in an endless loop. The loop from Line 3 to 5 termitates since the number of cells in a test case must be finite. The loop from Line 9 to 17 terminates since i is increased in every iteration and the number of cells in a spreadsheet must be finite. Another prerequisite for the termination of the algorithm is that the constraint solver terminates.

5 Empirical Evaluation

In this section, we evaluate ConBug by means of a spreadsheet corpus. This evaluation has three goals: (1) to empirically analyze the runtime behavior of ConBug, (2) to show the reduction that can be achieved when using ConBug, and (3) to compare ConBug with another spreadsheet fault localization approach. This section consists of three major parts: First, the solver used in this evaluation is presented. Afterwards, the used spreadsheet corpus is introduced. Finally, we evaluate ConBug with respect to runtime and reduction quality.

5.1 The used constraint solver

We developed a prototype for performing the empirical evaluation. This prototype uses Minion [19] as constraint solver. Minion is an out of the box, open source constraint solver and offers support for almost all arithmetic, relational, and logic operators such as multiplication, division, less, and equality over integers. For example, the multiplication $x * y = z$ is represented by `product(x,y,z)`.

Unlike other constraint solvers, Minion does not have to perform an intermediate transformation of the input constraint system. This allows for performance gains. As a trade off, the syntax of Minion requires a small overhead in modeling

[8] Known as iterative calculations; see `http://office.microsoft.com/en-us/excel-help/remove-or-allow-a-circular-reference-HP010066243.aspx`

the constraints compared to other constraint solvers: complex constraints have to be split into two or more simpler constraints.

MINION does not directly offer constraints for representing additions and subtractions. Therefore, we use the two relations `sumleq([x,y],z)` and `sumgeq([x,y],z)` (stating $x + y \leq z$ and $x + y \geq z$, respectively) for modeling the sum $x + y = z$ and the relations `weightedsumleq([x,y],[1,-1],z)` and `weightedsumgeq([x,y],[1,-1],z)` for modeling the subtraction $x - y = z$. The operator `weightedsumleq(cv,v,z)` ensures that $cv \cdot v \leq z$, where $cv \cdot v$ is the scalar dot product of cv and v, and `weightedsumleq(cv,v,z)` ensures that $cv \cdot v \geq z$, where $cv \cdot v$ is the scalar dot product of cv and v. We demonstrate this modeling overhead by means of a small example.

Example 11 *The expression* `A1 + B2 - C2` *is converted to the following* MINION *code:*

```
MINION 3
**VARIABLES**
DISCRETE A1{-2000..5000}
DISCRETE B2{-2000..5000}
DISCRETE C2{-2000..5000}
DISCRETE aux1{-2000..5000}
DISCRETE aux2{-2000..5000}
**CONSTRAINTS**
sumleq([A1,B2],aux1)
sumgeq([A1,B2],aux1)
weightedsumleq([1,-1],[aux1,C2], aux2)
weightedsumgeq([1,-1],[aux1,C2], aux2).
```

`aux1` *and* `aux2` *represent auxiliary variables that were introduced during conversion. The final result is stored in* `aux2`.

Minion does not offer a constraint for modeling `x<y` and `x>y`. However, these logic expressions can be modeled by using the constraint `ineq(x, y, k)` which ensures that $x \leq y + k$. The expression `x<y` can be modeled using the constraint `ineq(x,y,-1)` and `x>y` can be modeled using `ineq(y,x,-1)`. The constraint `ineq(x,y,-1)` ensures that x is smaller than y. However, if we want to know if x is smaller than y, we have to use MINION's reification mechanism: The constraint `reify(constraint, r)` ensures that the Boolean variable `r` is set to 1 if and only if the given constraint is satisfied. Therefore, the constraint `reify(ineq(x,y,-1), r)` can be used to answer the questions if x is smaller than y.

The function **IF** is also realized by using MINION's reification mechanism: The constraint `reifyimply(constraint, r)` ensures that the constraint is satisfied when r is set to 1. In contrast to the `reify` constraint, `reifyimply` allows that the constraint is satisfied while r is set to 0. Assuming that `aux1`, `aux2`, `aux3` are the auxiliary variables that are returned when evaluating `e1`, `e2`, `e3` and that `aux4` is a Boolean auxiliary variable and `aux5` is the variable that is returned, the function `if(e1;e2;e3)` can be modeled using the following constraints:

```
diseq(aux1,aux4)
reifyimply(eq(aux5,aux2),aux1)
reifyimply(eq(aux5,aux3),aux4)
```

Lets consider a small example using the logic expression $x > y$ and the **IF** function:

Example 12 *The expression* `IF(A1>A2;A1;A2)` *is converted to the following* MINION *code:*

```
MINION 3
**VARIABLES**
BOOL aux1
BOOL aux4
BOOL aux5
DISCRETE A1{-2000..5000}
DISCRETE A2{-2000..5000}
**CONSTRAINTS**
diseq(aux1,aux4)
reify(ineq(A2,A1,-1),aux1)
reifyimply(eq(aux5,A1),aux1)
reifyimply(eq(aux5,A2),aux4)
```

`aux1`, `aux4`, *and* `aux5` *are auxiliary variables introduced during the conversion. Algorithm* CONVERTEXPRESSION *(Algorithm 1) would return* `aux5` *as variable.*

The modeling of the 'health' status of a cell c ($\texttt{AB(c)} \vee \texttt{con}_\texttt{c}$) can be modeled using MINION's `watched-or(C1,...,Cn)` constraint: This constraint ensures that at least one of the constraints `C1`, ..., `Cn` is satisfied. Therefore, $\texttt{AB(c)} \vee \texttt{con}_\texttt{c}$ can be modeled as $\texttt{watched-or}(\texttt{eq}(\texttt{ab}_\texttt{c}, 1), \texttt{con}_\texttt{c})$. Having now discussed the basics of MINION, we could now show the complete MINION code for our running example.

Example 13 *Algorithm* CONBUG *(Algorithm 3) produces the following* MINION *code for our running example from Figure 1(c):*

```
MINION 3
**VARIABLES**
DISCRETE B2 {-2000..5000}
...
DISCRETE D4 {-2000..5000}
BOOL ab[7]
**SEARCH**
VARORDER [ab]
PRINT ALL
**CONSTRAINTS**
watched-or({element(ab,0,1), sumleq(B2,D2)})
watched-or({element(ab,0,1), sumgeq(B2,D2)})
watched-or({element(ab,1,1), product(D2,E2,F2)})
watched-or({element(ab,2,1), sumleq([B3,C3],D3)})
watched-or({element(ab,2,1), sumgeq([B3,C3],D3)})
watched-or({element(ab,3,1), product(D3,E3,F3)})
watched-or({element(ab,4,1), sumleq([B2,B3],B4)})
watched-or({element(ab,4,1), sumgeq([B2,B3],B4)})
watched-or({element(ab,5,1), sumleq([C2,C3],C4)})
watched-or({element(ab,5,1), sumgeq([C2,C3],C4)})
watched-or({element(ab,6,1), sumleq([D2,D3],D4)})
watched-or({element(ab,6,1), sumgeq([D2,D3],D4)})
#TEST CASE
eq(B2,23)
eq(C2,31)
eq(B3,35)
eq(C3,34)
eq(E2,15)
eq(E3,17)
eq(F2,810)
```

```
eq(F3,1173)
eq(B4,58)
eq(C4,65)
eq(D4,123)
#Solution size
watchsumgeq(ab,1)
watchsumleq(ab,1)
**EOF**
```

The MINION *code consists of three major parts: (1) the definition of the used variables and their domains (V_Π and D), (2) the constraints of the spreadsheet Π (CON_Π) and (3) the constraints of the test case T (CON_{TC}). The term* `VARORDER [ab]` *is used (1) to influence the search order for performance reasons and (2) to avoid multiple reports of the same variable assignment of* `ab`*. In this example code, we are interested in diagnoses of size 1 (see* `watchsumgeq(ab,1),watchsumleq(ab,1)`*).*

5.2 The spreadsheet corpus

For the empirical evaluation, we created a spreadsheet collection containing spreadsheets with integer values only, since the MINION constraint solver only supports Integer and Boolean variables. This Integer spreadsheet corpus is publicly available[9]. Our spreadsheet collection consists of both, artificially created spreadsheets, and real-life spreadsheets. We automatically created faulty versions for the spreadsheets by randomly modifying formulae. This process, also known as spreadsheet mutation, is known to yield representative faults [4]. We mutated formula cells randomly by

- replacing cell references with other references or constant values,
- replacing functions with constant values,
- changing the range of areas within functions, and
- changing arithmetic and relational operators.

In total, this corpus consists of 78 faulty spreadsheets. These spreadsheets are very different concerning their nature: while some have an economic background (e.g. a spreadsheet for computing the amortization time of an investment), others are used for mathematical computations (e.g. the Euclidean algorithm or the Calculation of the Fibonacci numbers), sport events (e.g. the computation of the winner of a championship), grading in schools or private issues (e.g. a calculator for the price of a new bedroom). On average, the spreadsheets contain 27.5 formulas. The smallest spreadsheet contains 7 formulas, while the largest spreadsheet contains 81 formulas. For each spreadsheet, the number of formula cells is indicated in the Tables 1 and 2. From these 78 spreadsheets, 28 spreadsheets contain only single faults, while 39 spreadsheets contain double and 11 spreadsheets triple faults (i.e. three cells have wrong formulas).

5.3 Results

The evaluation was performed on an Intel Core2 Duo processor (2.67 GHz) with 4 GB RAM and Windows 7 as operating system. We used the MINION version 0.15.

[9] `https://dl.dropbox.com/u/38372651/Spreadsheets/Integer_Spreadsheets.zip`

The computation time is the average time over 100 runs. We only computed the diagnoses with lowest cardinality, i.e. we only computed double fault diagnoses when MINION did not report any single fault diagnoses.

We measured the diagnosis quality by means of the achieved reduction and the time required for computing the diagnoses. To reduce any potential noise of the underlying algorithms and/or environment, the computation time is the average of 100 runs. Tables 1 and 2 show the results. The column 'Formula cells' indicates the number of cells containing a formula. The column 'Cells in diagnoses' indicates the number of cells that are contained in any diagnosis. These numbers are used to compute the 'Reduction' quality:

$$Reduction = \big(1 - \frac{Cells\ in\ diagnoses}{Formula\ cells}\big) \times 100\,\%. \tag{6}$$

On average, CONBUG reduces the number of formula cells that need to be manually inspected by 57.5 %. This result indicates that CONBUG has the potential to aid users while debugging faulty spreadsheets. The reduction of the number of cells in the diagnostic report returned to the user is an improvement over other approaches, such as spectrum-based fault localization, because it recudes the amount of information (diagnosis candidates) shown to the user. The column 'Constraints' indicates the number of constraints contained in the CSP. An observation is that there is not exist a correlation between the number of formula cells and the number of constraints. Two reasons for this observation are:

(1) A complex formula is translated into several constraints.
(2) Not all formula cells are translated into constraints.

The column 'Computation time' indicates the time, in milliseconds, required to solve the CSP. On average, the computation of all diagnoses with the lowest cardinality requires 1.1 seconds. Further, we observe that the number of constraints is not a good predictor for the required solving time. For example, MINION requires more than 33 seconds to solve the CSP of the spreadsheet 'cake_3_1' consisting of 53 constraints. In contrast, MINION requires less than one second to solve the CSP of the spreadsheet 'shopping_bedroom1_2_3'. Hence, more than the number of constraints, what impacts the time is the topological and formulae complexity of the spreadsheets. The average computation time of 1.1 seconds makes CONBUG applicable as a real-time approach.

Figure 2 illustrates the reduction quality with respect to the spreadsheets evaluated in the Tables 1 and 2. This histogram shows that only for 10 spreadsheets CONBUG is not able to significantly reduce the number of cells that have to be manually investigated (reduction between 0 % and 10 %). For 17 spreadsheets, the percentage of formula cells that have to be manually investigated could be reduced by more than 95 %. The good/poor performance is explained by the topology of the spreadsheets: spreadsheets that split complex formulae into several cells yield better results.

Table 3 and Figure 3 show a comparison of CONBUG and spectrum-based fault localization (SFL) [24,25] using the Ochiai coefficient. This comparison consists of two parts: First, we compare the average maximal effort that must spend. We define the maximal effort by the number of cells that have to be inspected at most in order to identify the faulty cell(s). We consider the average maximal effort in total and separately for single, double and triple faults. For all fault types, the

Table 1 Results of the empirical evaluation - Part 1

Spreadsheet	Formula cells	Cells in diagnoses	Reduction (%)	Constraints	Time (ms)
amortization_1_1	16	15	6.3	16	62
amortization_2_1	16	13	18.8	16	52
amortization_2_2	16	13	18.8	16	51
amortization_2_3	16	10	37.5	16	66
area_2_1	81	30	63.0	17	1047
area_2_2	81	28	65.4	22	947
area_2_3	81	59	27.2	22	1033
arithmetics00_1_1	8	7	12.5	22	105
arithmetics00_1_3	8	5	37.5	23	87
arithmetics00_2_2	8	8	0.0	23	174
arithmetics00_2_3	8	4	50.0	28	110
arithmetics00_3_1	8	8	0.0	28	169
arithmetics01_1_1	11	1	90.9	28	9
arithmetics01_1_2	11	4	63.6	34	10
arithmetics01_1_3	11	11	0.0	34	586
arithmetics02_2_2	16	5	68.8	34	343
arithmetics02_2_3	16	14	12.5	34	4001
arithmetics02_3_1	16	13	18.8	34	1936
austrian_league_1_1	32	1	96.9	34	196
austrian_league_1_2	32	1	96.9	34	280
austrian_league_2_1	32	7	78.1	34	734
austrian_league_2_2	32	1	96.9	34	215
austrian_league_2_3	32	1	96.9	34	162
austrian_league_3_1	32	24	25.0	36	1929
birthdays_1_1	39	9	76.9	42	194
birthdays_1_3	39	1	97.4	43	81
birthdays_3_1	39	4	89.7	52	163
cake_1_1	69	44	36.2	52	7350
cake_2_1	69	43	37.7	52	7315
cake_2_2	69	20	71.0	52	7011
cake_2_3	69	17	75.4	53	6913
cake_3_1	69	35	49.3	53	33123
computer_shopping_1_1	36	1	97.2	53	138
computer_shopping_1_2	36	1	97.2	68	138
computer_shopping_2_1	36	2	94.4	68	261
computer_shopping_2_2	36	1	97.2	68	140
computer_shopping_2_3	36	34	5.6	69	628
computer_shopping_3_1	36	1	97.2	69	142
conditionals01_1_1	11	2	81.8	69	24
conditionals01_1_2	11	7	36.4	69	32
conditionals01_2_1	11	4	63.6	87	29
conditionals01_2_2	11	5	54.5	87	33
conditionals01_2_3	11	7	36.4	87	34
conditionals02_1_1	7	3	57.1	101	50
conditionals02_1_3	7	3	57.1	101	79
conditionals02_2_1	7	3	57.1	101	76
conditionals02_2_2	7	3	57.1	101	77
conditionals02_2_3	7	4	42.9	102	71
conditionals02_3_1	7	3	57.1	102	68
dice_rolling_1_1	21	6	71.4	159	205
dice_rolling_2_1	21	7	66.7	159	231
dice_rolling_2_2	21	6	71.4	159	232
dice_rolling_2_3	21	6	71.4	159	205
dice_rolling_3_1	21	7	66.7	160	231

Table 2 Results of the empirical evaluation - Part 2

Spreadsheet	Formula cells	Cells in diagnoses	Reduction (%)	Constraints	Time (ms)
matrix_1_1	13	6	53.8	160	37
matrix_1_2	13	7	46.2	160	35
matrix_2_1	13	1	92.3	161	31
matrix_2_2	13	1	92.3	188	31
matrix_2_3	13	6	53.8	190	31
matrix_3_1	13	1	92.3	195	32
prom_calculator_1_1	14	13	7.1	195	23
prom_calculator_2_1	14	13	7.1	195	88
prom_calculator_2_2	14	13	7.1	195	18
prom_calculator_2_3	14	13	7.1	195	16
prom_calculator_3_1	14	10	28.6	196	21
shares_1_1	39	1	97.4	207	455
shares_1_2	39	1	97.4	207	388
shares_1_3	39	1	97.4	207	402
shares_1_4	39	1	97.4	207	328
shares_1_5	39	1	97.4	209	387
shares_2_2	39	2	94.9	210	1190
shares_2_3	39	18	53.8	210	2456
shopping_bedroom1_1_2	32	15	53.1	211	154
shopping_bedroom1_2_1	32	15	53.1	267	163
shopping_bedroom1_2_2	32	15	53.1	268	99
shopping_bedroom1_2_3	32	16	50.0	302	93
shopping_bedroom1_3_1	32	31	3.1	302	803
shopping_bedroom2_1_2	64	1	98.4	302	246
Average	**27.5**	**9.9**	**57.5**	**107.6**	**1,116.7**
Median	21.0	6.0	57.1	69.0	158.0
Stdev	19.9	11.4	31.9	82.6	4,016.3

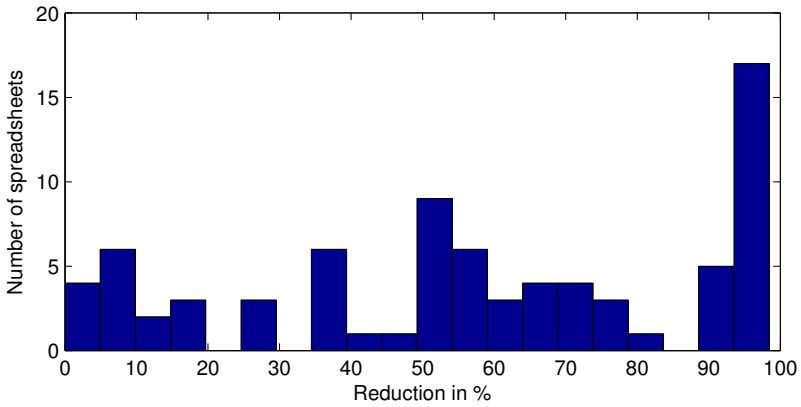

Fig. 2 Reduction histogram for the faults spreadsheets from Tables 1 and 2

effort is smaller when using ConBug. On average over all spreadsheets, the faulty cell(s) can be found when investigating 9.9 cells at most when using ConBug and 13.5 cells when using SFL. Second, we investigate how often ConBug is better than SFL and vice versa. ConBug yields better results for 30 spreadsheets, as shown in the dots below the identity line in Figure 3, while SFL yields better results for 12 spreadsheets. For 36 spreadsheets, ConBug and SFL yield results of the same size, and are shown in Figure 3 on top of the identity line.

Table 3 Comparison of ConBug and SFL with respect to (1) the average maximal effort until a fault is localized and (2) how often one approach is better than the other.

Fault type	**Average effort**		**Number of better rankings**		
	SFL	**ConBug**	**SFL**	**ConBug**	**equal**
Single	7.6	6.0	2	8	18
Double	18.4	12.0	6	17	16
Triple	13.5	12.5	4	5	2
Average / Total	**13.8**	**9.9**	**12**	**30**	**36**

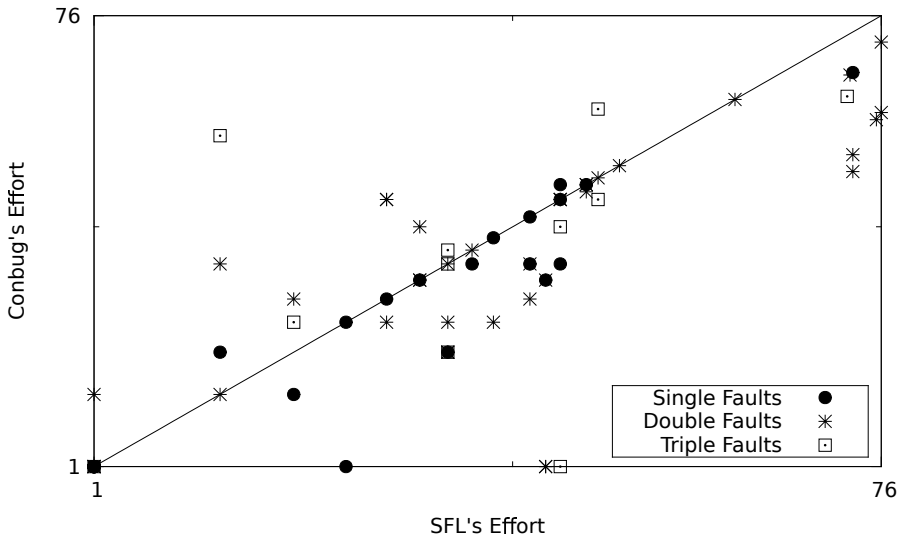

Fig. 3 ConBug vs. SFL's Effort

There exist two other approaches that deal with debugging of spreadsheets: GoalDebug [2] and the work of Jannach and Engler [26]. Unfortunately, it is not possible to compare ConBug with these approaches for the following reasons: (1) Neither the tool GoalDebug nor the spreadsheets used to evaluate GoalDebug are available [5]. (2) The approach of Jannach and Engler requires more information about the spreadsheets, as in contrast to our work, they rely on more test cases. For comparing their work with ConBug, one would have to create more test

cases. As the performance of their approach highly depends on the quality of the test cases and there is no spreadsheet corpus available that contains more than one test case, we refrain from comparing to them. Despite the fact that there are techniques to automatically generate test cases for spreadsheets, e.g., [17], we argue that end-users may not have more data than the one inserted into the spreadsheet. This scenario hinders the practical applicability of the approach, when compared to the one proposed in this paper.

5.4 Threats to Validity

A threat to external validity is that the underlying constraint solver poses a type-related limitation: the prototype only accepts spreadsheets containing integers. Real-world spreadsheets contain other data types such as real-valued or float data types. Therefore, it is possible that the results for a different set of spreadsheets with different characteristics may produce different results. However, this is a technological limitation of the used solver (we considered it because the authors were familiar with it), but it could be replaced by other solvers to handle other data types.

The construct validity threat is that the performance of ConBug was evaluated using a metric that measures diagnostic effort in terms of the number of cells that one needs to inspect (i.e., that have been indicted by ConBug). This metric assumes that only those cells are inspected. In practice, that may not be the case as other cells might be inspected because they are related to indicted cells.

The internal validity threat is that eventual faults in the implementation of ConBug or in the underlying constraint solver MINION may invalidate the results. To mitigate this threat, we have not only thoroughly tested our scripts but also manually checked a large set of results.

6 Related Work

The work presented in this paper is based on model-based diagnosis [34], namely its application to (semi-)automatic debugging (e.g., [10]). In contrast to previous work, the work presented in this paper does not use logic-based models of programs but, instead, a constraint representation and a general constraint solver. A similar approach to the one of this paper has been presented in [43] to aid debuggers in pinpointing software failures.

Jannach and Engler [26] also presented a model-based approach for the debugging of spreadsheets. Their approach which is part of the Exquiste framework is based on an extended hitting-set algorithm and user-specified or historical test cases and assertions, to calculate possible error causes in spreadsheets. While the general idea of their approach is similar to ConBug, the technical realization is slightly different: (1) While Jannanch and Engler use a Hitting-Set Algorithm [34], we encode the reasoning about the correctness of individual formulas directly into the constraint representation. Studies [29] have shown that the latter approach yields performance gains. (2) We rely on a single test case while Jannach and Engler require several test cases for their approach.

GoalDebug [2] is a spreadsheet debugger for end users. Whenever the computed output of a cell is incorrect, the user can supply an expected value for a cell, which is employed by the system to generate a list of change suggestions for formulae that, when applied, would result in the user-specified output. In [2], a thorough evaluation of the tool is stated. GoalDebug employs a distinct constraint-based approach from the one presented in this paper. Unlike ConBug, GoalDebug relies upon a set of possible, pre-defined change (repair) inference rules. The fault localization approach is done by mutating the spreadsheet using the set of rules and ascertain that the user expectations are met. Unless the set of changes is general enough, GoalDebug does not generalize as much as ConBug. ConBug, is more general in the sense that it delegates the constraint satisfaction problem to a general purpose, off-the-shelf constraint solver. Moreover, relying on the pre-defined set of repairs, it suggests a list of changes to fix the spreadsheet (which is not currently supported by ConBug).

Another line of research in the spreadsheet domain focuses on trace-based fault localization. Reichwein et al. [33,37] adapted the concept of program slicing to spreadsheets. In their approach, they rely on user-specified information about correct and incorrect cell values. Cells that contribute to erroneous cell values are more likely to be faulty then cells contributing to correct cell values. A similar technique was proposed by Ayalew and Mittermeir [12]. Their data-flow driven approach prioritizes cells based on the number of incorrect successor cells and predecessor cells. While the previously mentioned techniques rely on the principles of spectrum-based fault localization, Hofer et al. [24,25] were the first who explicitly adapted the concepts of spectrum-based fault localization from the software to the spreadsheet domain.

Spreadsheet testing is closely related to debugging. In the WYSIWYT system users can indicate incorrect output values by placing a faulty token in the cell. Similarly, they can indicate that the value in a cell is correct by placing a correct token [35]. When a user indicates one or more program failures during this testing process, fault localization techniques [36] direct the user's attention to the possible faulty cells. Similar to our approach, WYSIWYT provides no help with regard to how to change erroneous formulae. In contrast to ConBug, WYSIWYT also collects user input about correct cell values and employs this information in the fault localization analysis.

There are several spreadsheet analysis tools that try to reason about the units of cells to find inconsistencies in formulae, e.g., [1,11]. The tools differ in the rules they employ and in the degree to which they require users to provide additional input. Most of these approaches require the user the annotate the spreadsheet cells with additional information, except the UCheck system [3], which can perform unit analysis automatically by exploiting techniques for automated header inference [1]. However, none of these approaches provide any further help to the user to correct the errors once they are detected. Other approaches aimed at minimizing the occurrence of errors in spreadsheets include code inspection [31] and adoption of better spreadsheet design practices [16]. However, none of these approaches focus on debugging of spreadsheets.

7 Conclusions and Future Work

In this paper, we propose ConBug, a constraint-based approach for automatically debugging spreadsheets. The approach takes as input a spreadsheet and the set of user expectations (specifying the input and output cells and their expected values), and produces as output a set of diagnosis candidates. Diagnosis candidates are explanations for the misbehavior in user expectations. Our empirical investigation shows that ConBug is light-weight and efficient.

This line of research raises a number of research questions that require further investigation. First and foremost, our intention is to release the approach in a plug-in for spreadsheet applications. As such, and keeping in mind that the target audience are end-users, we plan the devise a natural, intuitive way to visually display the diagnostic information. Second, we plan to combine this work with mutation of spreadsheets [4] to be able to give advice to users on how to fix the buggy spreadsheet. Third, we plan to study the applicability and efficiency of other, more light-weight techniques to debug spreadsheets. In particular, we will study the complexity-efficiency trade-off using spectrum-based reasoning for fault localization [10], which is amongst the best approaches for software fault localization. Fourth, currently our approach is limited to the integer domain, we plan to extend our approach to be able to handle, e.g., strings and floats.

Acknowledgment

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

References

1. Abraham, R., Erwig, M.: Header and unit inference for spreadsheets through spatial analyses. In: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing, VLHCC '04, pp. 165–172. IEEE Computer Society, Washington, DC, USA (2004). DOI 10.1109/VLHCC.2004.29. URL http://dx.doi.org/10.1109/VLHCC.2004.29
2. Abraham, R., Erwig, M.: Goaldebug: A spreadsheet debugger for end users. In: Proceedings of the 29th international conference on Software Engineering, ICSE '07, pp. 251–260. IEEE Computer Society, Washington, DC, USA (2007). DOI 10.1109/ICSE.2007.39. URL http://dx.doi.org/10.1109/ICSE.2007.39
3. Abraham, R., Erwig, M.: Ucheck: A spreadsheet type checker for end users. Journal of Visual Languages and Computing **18**, 71–95 (2007). DOI 10.1016/j.jvlc.2006.06.001
4. Abraham, R., Erwig, M.: Mutation operators for spreadsheets. IEEE Transactions on Software Engineering **35**(1), 94–108 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/TSE.2008.73
5. Abraham, R., Erwig, M.: Personal communication (2013)
6. Abreu, R., Mayer, W., Stumptner, M., van Gemund, A.J.C.: Refining spectrum-based fault localization rankings. In: Proceedings of the 2009 ACM symposium on Applied Computing, SAC '09, pp. 409–414. ACM, New York, NY, USA (2009). DOI 10.1145/1529282.1529374. URL http://doi.acm.org/10.1145/1529282.1529374

7. Abreu, R., Riboira, A., Wotawa, F.: Constraint-based debugging of spreadsheets. In: Ibero-American Conference on Software Engineering (CibSE'12), pp. 1 – 14 (2012)
8. Abreu, R., Riboira, A., Wotawa, F.: Debugging of spreadsheets: A CSP-based approach. In: Third IEEE International Workshop on Program Debugging (2012)
9. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, TAICPART-MUTATION '07, pp. 89–98. IEEE Computer Society, Washington, DC, USA (2007). URL http://dl.acm.org/citation.cfm?id=1308173.1308264
10. Abreu, R., Zoeteweij, P., Gemund, A.J.C.v.: Spectrum-based multiple fault localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, pp. 88–99. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/ASE.2009.25. URL http://dx.doi.org/10.1109/ASE.2009.25
11. Ahmad, Y., Antoniu, T., Goldwater, S., Krishnamurthi, S.: A type system for statically detecting spreadsheet errors. In: 18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada, pp. 174–183. IEEE Computer Society (2003). DOI http://csdl.computer.org/comp/proceedings/ase/2003/2035/00/20350174abs.htm
12. Ayalew, Y., Mittermeir, R.: Spreadsheet debugging. Bilding Better Business Spreadsheets - from the ad-hoc to the quality-engineered. Proceedings of EuSpRIG 2003, Dublin, Ireland, July 24th-25th 2003 pp. 67–79 (2003)
13. Ceballos, R., Gasca, R.M., Borrego, D.: Constraint satisfaction techniques for diagnosing errors in design by contract software. SIGSOFT Softw. Eng. Notes **31**(2) (2005). DOI 10.1145/1108768.1123070. URL http://doi.acm.org/10.1145/1108768.1123070
14. 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), 133–143 (2001). DOI 10.1023/A:1016631003750. URL http://dx.doi.org/10.1023/A:1016631003750
15. Collavizza, H., Rueher, M.: Exploring different constraint-based modelings for program verification. In: Proceedings of the 13th international conference on Principles and practice of constraint programming, CP'07, pp. 49–63. Springer-Verlag, Berlin, Heidelberg (2007). URL http://dl.acm.org/citation.cfm?id=1771668.1771676
16. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC '10, pp. 93–100. IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/VLHCC.2010.22. URL http://dx.doi.org/10.1109/VLHCC.2010.22
17. Fisher, M., Cao, M., Rothermel, G., Cook, C.R., Burnett, M.M.: Automated test case generation for spreadsheets. In: Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on, pp. 141–151. IEEE (2002)
18. Fisher, M., Rothermel, G.: The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. SIGSOFT Softw. Eng. Notes **30**(4), 1–5 (2005)
19. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast, scalable, constraint solver. In: Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva del Garda, Italy, pp. 98–102. IOS Press, Amsterdam, The Netherlands, The Netherlands (2006). URL http://dl.acm.org/citation.cfm?id=1567016.1567043
20. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, ISSTA '98, pp. 53–62. ACM, New York, NY, USA (1998). DOI 10.1145/271771.271790. URL http://doi.acm.org/10.1145/271771.271790
21. Gotlieb, A., Botella, B., Rueher, M.: A clp framework for computing structural test data. In: Proceedings of the First International Conference on Computational Logic, CL '00, pp. 399–413. Springer-Verlag, London, UK, UK (2000). URL http://dl.acm.org/citation.cfm?id=647482.728291
22. Hermans, F., Pinzger, M., van Deursen, A.: Supporting professional spreadsheet users by generating leveled dataflow diagrams. In: Proceeding of the 33rd international conference on Software engineering, pp. 451–460. ACM (2011)
23. Hermans, F., Sedee, B., Pinzger, M., van Deursen, A., Cheng, B., Pohl, K.: Data clone detection and visualization in spreadsheets. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, IEEE Computer Society (2013)

24. Hofer, B., Perez, A., Abreu, R., Wotawa, F.: On the empirical evaluation of similarity coefficients for spreadsheets fault localization. Automated Software Engineering pp. 1–28 (2014)
25. Hofer, B., Riboira, A., Wotawa, F., Abreu, R., Getzner, E.: On the empirical evaluation of fault localization techniques for spreadsheets. In: V. Cortellessa, D. Varró (eds.) Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013., *Lecture Notes in Computer Science*, vol. 7793, pp. 68–82. Springer (2013)
26. 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, pp. 252–264. Kaunas, Lithuania (2010)
27. 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), 21:1–21:44 (2011). DOI 10.1145/1922649.1922658. URL `http://doi.acm.org/10.1145/1922649.1922658`
28. Mayer, W.: Static and hybrid analysis in model-based debugging. Ph.D. thesis, School of Computer and Information Science, University of South Australia (2007)
29. Nica, I., Pill, I., Quaritsch, T., Wotawa, F.: The route to success - a performance comparison of diagnosis algorithms. In: F. Rossi (ed.) IJCAI. IJCAI/AAAI (2013)
30. Nica, M., Nica, S., Wotawa, F.: On the use of mutations and testing for debugging. Software : Practice & Experience (2012)
31. Panko, R.R.: Applying code inspection to spreadsheet testing. Journal of Management Information Systems **16**, 159–176 (1999)
32. Peischl, B., Wotawa, F.: Automated source-level error localization in hardware designs. IEEE Design Test of Computers **23**, 8–19 (2006). DOI 10.1109/MDT.2006.5
33. Reichwein, J., Rothermel, G., Burnett, M.: 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. Austin, Texas (1999)
34. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1), 57–95 (1987)
35. 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, pp. 230–239. ACM, New York, NY, USA (2000). DOI 10.1145/337180.337206. URL `http://doi.acm.org/10.1145/337180.337206`
36. 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, pp. 123–132. ACM, New York, NY, USA (2003). DOI 10.1145/774833.774851. URL `http://doi.acm.org/10.1145/774833.774851`
37. Ruthruff, J.R., Prabhakararao, S., Reichwein, J., Cook, C., Creswick, E., Burnett, M.: Interactive, Visual Fault Localization Support for End-User Programmers. Journal of Visual Languages & Computing **16**(1-2), 3–40 (2005)
38. Tukiainen, M.: Uncovering effects of programming paradigms: Errors in two spreadsheet systems. In: Proc. PPIG'00, pp. 247–266 (2000)
39. 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, pp. 364–374. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/ICSE.2009.5070536. URL `http://dx.doi.org/10.1109/ICSE.2009.5070536`
40. Woods, S., Yang, Q.: Program understanding as constraint satisfaction: Representation and reasoning techniques. Automated Software Engg. **5**(2), 147–181 (1998). DOI 10.1023/A:1008655230736. URL `http://dx.doi.org/10.1023/A:1008655230736`
41. Wotawa, F., Nica, M.: On the compilation of programs into their equivalent constraint representation. Informatica Journal **32**, 359–371 (2008)
42. Wotawa, F., Nica, M., Moraru, I.D.: Automated debugging based on a constraint model of the program and a test case. The journal of logic and algebraic programming **81**(4) (2012)
43. Wotawa, F., Weber, J., Nica, M., Ceballos, R.: On the complexity of program debugging using constraints for modeling the program's syntax and semantics. In: Proceedings of the Current topics in artificial intelligence, and 13th conference on Spanish association for

artificial intelligence, CAEPIA'09, pp. 22–31. Springer-Verlag, Berlin, Heidelberg (2010). URL http://dl.acm.org/citation.cfm?id=1893496.1893500