Constraint-based Debugging of Spreadsheets*

Rui Abreu[1], André Riboira[1], and Franz Wotawa[2]

[1] Dept. of Informatics Engineering, University of Porto, Portugal
rui@computer.org, andre.riboira@fe.up.pt
[2] Institute for Software Technology, Graz University of Technology, Austria
wotawa@ist.tugraz.at

Abstract. Despite being staggeringly error prone, spreadsheets can be viewed as a highly flexible end-users programming environment. As a consequence, spreadsheets are widely adopted for decision making by end-users, and may have a serious economical impact for the business. Hence, approaches for aiding the process of pinpointing the faulty cells in a spreadsheet are of great value. In this paper we present a constrain-based approach for debugging spreadsheets. We coin the approach ConBug. Essentially, the approach takes as input a (faulty) spreadsheet and a test case that reveals the fault (a test case specifies values for the input cells as well as the expected values for the output cells) and computes a set of diagnosis candidates for the debugging problem we are trying to solve. To compute the set of diagnosis candidates we convert the spreadsheet and test case to a constraint satisfaction problem (CSP), modeled using the state-of-the-art constraint solver MINION. We use a case study, in particular using a spreadsheet taken from the well-known EUSES Spreadsheet Corpus, to better explain the different phases of the approach as well as to measure the efficiency of ConBug. We conclude that ConBug can be of added value for the end user in order to pinpoint faulty cells.

1 Introduction

Spreadsheet tools, such as Microsoft Excel[3], iWork's Numbers[4], and OpenOffice's Calc[5], can be viewed as programming environments for non-professional programmers [1]. These so-called "end-user" programmers vastly outnumber professional ones: the US Bureau of Labor and Statistics estimates that more than 55 million people will be using spreadsheets and databases at work on a daily basis by 2012 [1]. 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

* 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.

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

an alarmingly high rate [2, 3]. 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[6].

In the software engineering domain, constraints have been used for various purposes like verification [4], debugging [5, 6], program understanding [7] as well as testing [8, 9]. 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 use constraints obtained from the spreadsheets directly.

In this paper, we propose a constraint-based approach for debugging spreadsheets, dubbed ConBug. 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 (an example of a diagnosis candidate is cell B1 *and* cell C4 are faulty, i.e., explain the *faulty* observed value in, e.g., cell A100). We describe how the approach works and its efficiency using three in-vitro spreadsheets plus a real spreadsheet taken from the large EUSES Spreadsheet Corpus[7].

2 Basic definitions

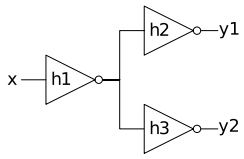
In order to be self contained, we briefly introduce the basic definitions that are relevant for this paper. The paper deals with fault diagnosis based on models of spreadsheets, i.e., an approach to (semi-) automatically pinpoint faulty cells in the spreadsheet is proposed. In this paper we assume a spreadsheet programming language $\mathcal{L}$ with syntax and semantics similar to, e.g., Microsoft Excel. Moreover, we assume correctness of standard functions ϕ provided by the spreadsheet (e.g., SUM, AVERAGE). In Figure 1, an example of a spreadsheet program is given as running example. The spreadsheet implements a 3-inverter circuit (see Figure 1(a)) with a defective cell, namely B5 (see Figure 1(b)).

In order to state the debugging problem, we assume a spreadsheet $\Pi \in \mathcal{L}$ containing (at least) a cell that does not behave as expected. In the context of this paper such a spreadsheet Π is faulty when there exist input values (cells) from which the spreadsheet computes output values (cells) differing from the expected values. The input and correct output values are provided to the spreadsheet by means of a test case. For defining test cases we introduce variable[8] environments (or environments for short). An environment is a set of pairs (x, v) where x is a variable and v its value. In an environment there is only one pair for a variable. We are now able to define test cases formally as follows.

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

[7] http://esquared.unl.edu/wikka.php?wakka=EUSESSpreadsheetCorpus

[8] In this paper we use the term variable and cell interchangeably.



(a) 3-inverter circuit

	A	B	C	D	E	F
1						
2	x	TRUE				
3	w	FALSE				
4	y1	TRUE				
5	y2	FALSE				
6						
7						
8						
9						
10						
11						

```
(y1,y2) 3inv(bool x) {
B3.   w = ¬x
B4.   y1 = ¬w;
B5.   y2 = w; //should be y2 = ¬w
      return (y1,y2);
    }
```

(b) A defective spreadsheet of the 3-inverter circuit

Fig. 1. Running Example: A faulty spreadhseet

Definition 1 (Input/Output cell). *An input cell is a cell that does have an influence on other cells of the spreadsheet. Conversely, an output cell is a cell that does not influence any other cell in the spreadhseet.*

Definition 2 (Test case). *A test case for a spreadsheet $\Pi \in \mathcal{L}$ is a tuple (I, O) where I is the input variable environment specifying the values of all input cells used in Π, and O the output variable environment (not necessarily specifying values for all output variables).*

For example a (failing) test case for the spreadsheet program from Figure 1 is $I_\Pi : \{B2 = TRUE\}$ and $O_\Pi : \{B4 = TRUE; B5 = TRUE\}$. This particular test case is the one depicted in the spreadsheet of Figure 1.

Definition 3 (Failing test case). *A test case is failing if there is at least one output cell that differs from the expected value.*

For the program from Figure 1 the test case (I_Π, O_Π) is a failing test case. For input I_Π the program returns $\{B4 = TRUE; B5 = FALSE\}$ which contradicts the expected output $O_\Pi : \{B4 = TRUE; B5 = TRUE\}$. Formally, we define passing and failing as follows:

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

Again, note that not all values have to be specified. However, it is necessary that all specified values for the output cells are returned as expected. A cell for which no value is specified in O can have an arbitrary value.

Definition 4 (Test suite). *A test suite TS for a spreadsheet $\Pi \in \mathcal{L}$ is a set of test cases of Π.*

A spreadsheet is said to be correct with respect to TS if and only if the program passes all test cases. Otherwise, we say that the program is incorrect or faulty.

If deemed incorrect, the faulty cells have to be found in order to fix the spreadsheet. The action of pinpointing the faulty locations is called debugging.

Definition 5 (Debugging problem). *Let $\Pi \in \mathcal{L}$ be a program and TS its test suite. If $T \in TS$ is a failing test case of Π, then (Π, T) is a debugging problem.*

A solution to the debugging problem is the identification and correction of a part of the spreadsheet (set of cells) responsible for the detected misbehavior. We call such a program part an explanation. There are many approaches that are capable of returning explanations including [10, 11, **?**,12–14] and [5, 15] among others. In this paper, we follow the debugging approach based on constraints, i.e., [5, 15]. In particular, the approach makes use of the program's constraint representation to compute possible fault candidates. So, debugging is reduced to solving the corresponding constraint satisfaction problem (CSP).

Definition 6 (Constraint Satisfaction Problem (CSP)). *A constraint satisfaction problem is a tuple (V, D, CO) 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 CO. A solution for a CSP represents a valid instantiation of the variables V with values from D such that none of the constraints from CO is violated.*

Note that the variables used in a CSP are not necessarily cells used in a spreadsheet. We discuss the representation of programs as a CSP in the next section. Afterwards we introduce an algorithm for computing diagnosis candidates given a CSP debugging problems. This algorithm only states cells as potential explanations for a failing test cases ; no information regarding how to correct the program is given.

3 CSP representation of spreadsheets

Before converting a spreadsheet $\Pi \in \mathcal{L}$ into its corresponding constraint representation, some intermediate transformation steps have to be performed. These transformations are necessary for removing any imperative constructs, i.e., making it a declarative one, as required by the constraint programming paradigm. Our algorithm for converting a program and encoding its debugging problem into a CSP comprises two main phases:

1. SSA conversion, and
2. constraint conversion.

SSA conversion $\Pi_{SSA} = SSA(\Pi)$ The static single assignment (SSA) form is an intermediate representation of a program with the property that no two left-side variable share the same name. This property of the SSA form allows for an easy conversion into a CSP. It is beyond our scope to detail the program-to-SSA conversion. However, to be self-contained we explain the necessary rules needed for converting spreadsheets into a SSA-like representation. For more details regarding the SSA-conversion of software programs see for example [6].

- We convert assignments by adding an index to a variable each time the variable is defined, i.e., occurs at the left side of an assignment. If a variable is re-defined, we increase its unique index by one such that the SSA-form property holds. The index of a referenced variable, i.e., a variable occurring at the right side of an assignment, equals to the index of the last definition of the variable.
- We split the conversion of conditional structures into three steps:
  (1) the entry condition is saved in an auxiliary variable,
  (2) each assignment statement is converted following the above rule, and
  (3) for each conditional statement and variable defined in the sub-block of the statement, we introduce an evaluation function

$$\Phi(\mathtt{v_{then}}, \mathtt{v_{else}}, \mathtt{cond}) \stackrel{\text{def}}{=} \begin{cases} \mathtt{v_{then}} \text{ if } \mathtt{cond} = true \\ \mathtt{v_{else}} \text{ otherwise} \end{cases}$$

  which returns the statement conditional-exit value, e.g., $\mathtt{v_{after}} = \Phi(\mathtt{v_{then}}, \mathtt{v_{else}}, \mathtt{cond})$.

  For example, the corresponding simplified form of a cell containing the fragment

$$\mathrm{A1} = \mathtt{IF(cond_{expr},\ \{E^1_{expr}\},\ \{E^2_{expr}\})}$$

  is given as follows:

```
cond_i = cond_expr;
A1_j = E^1_expr;
A1_k = E^2_expr;
A1_l = Φ(A1_j, A1_k, cond_i);
```

Constraint conversion $CON = CC(\Pi_{SSA})$ This last step of the conversion process transforms the statements to the corresponding constraints, including also the encoding of the debugging problem. For this purpose we introduce a special boolean variable $AB(S)$ for a cell S, that states the incorrectness of a cell S. The constraint model of a cell comprises corresponding constraints or-connected with $AB(S)$. Let $S \in \Pi_{SSA}$ and let C_S be the constraint encoding statement S in the constraint programing language. Note that ϕ functions cannot be incorrect. Hence, no AB variable is defined for statements using ϕ. We model S in CON as follows:

$$CON \cup \begin{cases} \mathtt{AB(S)} \vee \mathtt{C_S} & \text{if } S \text{ does not contain } \phi \\ \mathtt{C_S} & \text{otherwise} \end{cases}$$

Algorithm 1 Algorithm ComputeExpresion

Inputs: An expression $\texttt{E}_{\texttt{expr}}$ and an empty set M for storing the MINION constraints
Output: A set of representing the expression stored in M, and a variable or constant where the result of the conversion is finally stored

```
1 if E_expr is a variable or constant then
2     return E_expr
3 else
4     E_expr is of the form E^1_expr ψ E^2_expr
5     Let aux_1 = ComputeExpression (E^1_expr)
6     Let aux_2 = ComputeExpression (E^2_expr)
7     Generate a new MINON variable result and create MINON constraints accordingly to the given operator ψ, which define the relationship between aux_1, aux_2, and result, and add them to M
8     return result
9 end if
```

Hence the CSP representation of a program Π is given by the tuple

$$(V_{\Pi_{SSA}}, D_{SSA}, CON)$$

where $V_{\Pi_{SSA}}$ represents all variables of the SSA representation Π_{SSA} of program Π, defined over the domains $D_{SSA} = \{Integer, boolean\}$.

Once the SSA form of a spreadsheet is computed, what is then missing in the conversion process to the constraint satisfaction problem of the debugging problem. In our implementation we model the CSP to represent the debugging problem in the language of the MINION constraint solver [16]. MINION is an out of the box, open source constraint solver. Its syntax requires a little effort in modeling the constraints than other constraint solvers, e.g., it does not support different operators on the same constraint. Because of this drawback sometimes complex constraints have to be split into two or three more simpler constraints. However, because of this characteristic, MINION, unlike other constraint solver toolkits, does not have to perform an intermediate transformation of the input constraint system. MINION offers support for almost all arithmetics, relational, and logic operators such as minus, plus, multiplication, division, less, and equal over integers. Furthermore, it also requires that all expressions used in a MINION program to be limited to one operator.

Because of the syntactical limitations of MINON we have to convert an assignment statement with an expression $\texttt{E}_{\texttt{expr}}$ on the right-side comprising more than one operator into a sequence of MINON statements. The idea behind the conversion is straightforward. A constant or variable is represented by itself. For an expression of the form $\texttt{E}^1_{\texttt{expr}}\ \psi\ \texttt{E}^2_{\texttt{expr}}$ we convert $\texttt{E}^1_{\texttt{expr}}$ and $\texttt{E}^2_{\texttt{expr}}$ separately, and assign a new intermediate variable for each converted sub-expression. The ComputeExpression algorithm (see Algorithm 1) implements the conversion.

As an example, the expression `a_0 + b_0 - c_0` is converted to the following MINION constraints using ComputeExpression where `aux1` and `aux2` represent new variables introduced during conversion.

```
sumleq([a_0,b_0],aux1)
sumgeq([a_0,b_0],aux1)
weightedsumleq([1,-1],[aux1,c_0], aux2)
weightedsumgeq([1,-1],[aux1,c_0], aux2)
```

In this example the MINION constraints `sumleq` and `sumgeq` are used to represent the plus operator, and `weightedsumleq` and `weightedsumgeq` together with the given list of signs are for representing the minus operator. We summarize the conversion of the SSA representation of the spreadsheet to MINION constraints in Table 1.

Statement	**MINION Constraint**
A1 = exp;	auxVar = **ComputeExpression**(exp), *eq*(A1, auxVar)
A2 = (A1 > 0);	*reify*(*ineq*(0,A1,-1),A2)
A3 = A2 & (A1 > 0);	*reify*(*ineq*(0,A1,-1),cond_aux) *reify*(*watchsumgeq*([A2,cond_aux], 2),A3)
A4 = Φ(A5, A7, A2);	*watched-or*(*eq*(A2,0), eq(A4,A5)) *watched-or*(*eq*(A2,1), eq(A4,A7))

Table 1. MINION constraints conversion

For convenience we assume a function CONVERT that implements the conversion of spreadsheets into MINION constraints as discussed in this section. Hence, CONVERT takes the spreadsheet as input and returns a set of MINION constraints as output. We use this function in the next section, where we discuss an algorithm for debugging spreadsheets using contraints.

4 Debugging

Debugging of a spreadhseet requires the existence of a failing test case. This means that in addition to the set of constraints CON, we must add an extra set of constraint encoding a failing test case (I, O). For all $(x, v) \in I$ the constraint $x\_0 = v$ is added to the constraint system. For all $(y, w) \in O$ the constraint $y\_\iota = w$ is added where ι is the greatest index of cell y in the SSA form. Let CON_{TC} denote the constraints resulted from converting the given test case. Then, the CSP corresponding to the debugging problem of a program Π is now represented by the tuple

$$(V_{\Pi_{SSA}}, D_{SSA}, CON \cup CON_{TC})$$

Again, for convenience, we assume a function CONVERT_TEST that implements the conversion of the failig test case into MINION constraints as outlined. Hence, CONVERT_TEST takes the the failing test case as input and returns a set of MINION constraints as output.

Algorithm 2 Algorithm ConBug

Inputs: A spreadsheet Π and a failing test case T
Output: Diagnostic Report D

```
1  D ← ∅
2  CON_Π ← convert(Π)
3  CON_T ← convert_test(T)
4  i ← 1
5  while i ≤ Cells(Π) do
6      D ← Minion(CON_Π, CON_T, i)
7      if D ≠ ∅ then
8          return D
9      else
10         i ← i + 1
11     end if
12 end while
13 return D
```

Let CON_Π be the constraint representation of a spreadsheet Π and CON_T the constraint representation of a failing test case T. The debugging problem formulated as a CSP comprises CON_Π together with CON_T. Note that in CON_Π assumptions about correctness or incorrectness of cells are given, which are represented by a variable AB assigned to each statement. The algorithm for computing bug candidates calls the MINION CSP solver using the constraints and asks for a return value of AB as a solution. The size (cardinality) of the solution corresponds to the size of the bug, i.e., the number of statements that must be changed together in order to explain the misbehavior. 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 and iterates calling the constraint solver. This is done until either a solution is found or the maximum size of a bug, which is equivalent to the number of statements in Π, is reached.

In summary, the automatic fault localization approach proposed in the paper comprises 3 main phases. The first phase comprises the conversion of a spreadsheet $\Pi \in \mathcal{L}$ into the corresponding set of MINION constraints (line 2 in Algorithm 2). The second phase is the conversion of the failing test case into the corresponding set of MINION constraints (line 3 in Algorithm 2). Finally, the third phase comprises the computation of diagnosis candidates, i.e., cells of the spreadsheet that might cause the revealed misbehavior, from the constraint representation of a spreadsheet $\Pi \in \mathcal{L}$ (lines 4 to 12 in Algorithm 2). Eventually, the algorithm returns the empty set if no diagnosis candidates are found (i.e., no solution is found for the CSP problem).

5 Case Study

This section details how the approach works using four different faulty spreadsheets. The first case is a spreadsheet that represents the inversor problem introduced before. The second case is a spreadsheet that represents an adaptation from the example used in [17], which describes an automatic approach for software debugging. The third case is a sample spreadsheet that mimics a common user made spreadsheet, with a faulty formula to calculate the cardiac output of a human. Finally, the fourth case is a spreadsheet from EUSES Spreadsheet Corpus modified to have a faulty cell.

The first spreadsheet is converted into the following MINION model (the spreadsheet itself is presented in Section 2):

```
MINION 3
**VARIABLES**
BOOL b2
BOOL w
BOOL b4
BOOL b5
BOOL ab[3]
**SEARCH**
VARORDER [ab]
PRINT ALL
**CONSTRAINTS**
watched-or({element(ab, 0, 1), diseq(w, b2)})
watched-or({element(ab, 1, 1), diseq(b4, w)})
watched-or({element(ab, 2, 1), eq(b5, w)})
#TEST CASE
eq(b2,1)
eq(b4,1)
eq(b5,1)
#SD
watchsumgeq(ab, 1)
watchsumleq(ab, 3)
**EOF**
```

Executing the MINION solver with such model yields one diagnosis candidate (cell B5 is faulty). Thus, ConBug points out that there is just one solution for the model, solution that represents the faulty cell for this specific spreadsheet and test case.

The second case study is the adaptation to a spreadsheet of the example used in [17] (see Figure 2). The spreadsheet is modeled using the following constrains:

```
**CONSTRAINTS**
watched-or({element(ab,0,1), product(2,b1,b4)})
watched-or({element(ab,1,1), product(2,b2,b5)})
watched-or({element(ab,2,1), sumgeq([b4,b5],b7)})
watched-or({element(ab,2,1), sumleq([b4,b5],b7)})
watched-or({element(ab,3,1), product(b4,b4,b8)})
#TEST CASE
```

	A	B	C
1	X value	1	
2	Y value	2	
3			
4	i value	2	---> (2*B1)
5	j value	4	---> (2*B2)
6			
7	o1	**6**	---> (B4+B5)
8	o2	**4**	---> (B4*B4)

Fig. 2. Example Spreadsheet with a traditional software port

```
eq(b1,1)
eq(b2,2)
eq(b7,8)
eq(b8,4)
```

And the solver identifies one solution for the model, i.e., identifies the potential faulty cell (B7, third variable of the array): As in [17] (which used a similar problem as a software program), our approach properly identifies the faulty cell.

The third case study is a spreadsheet that tries to mimic traditional end-user made spreadsheets. This spreadsheet calculates the Cardiac Output based on the input of three values and two formulas (see Figure 3).

	A	B	C	D	E	F
1	**Cardiac Output (Q) Calculation**					
2						
3	**Inputs**					
4	End Diastolic Volume (EDV)	**120**	**ml**			
5	End Systolic Volume (ESV)	**50**	**ml**			
6	Heart Rate (HR)	**72**	**bpm**			
7						
8	**Calculations**					
9	Stroke Volume (SV)	48	ml			
10	Cardiac Output (Q)	**3456**	**ml / minute**			
11						
12	**Equations**					
13	Cardiac Output (Q) = Stroke Volume (SV) * Heart Rate (HR)					
14	Stroke Volume (SV) = End Diastolic Volume (EDV) – End Systolic Volume (ESV)					

Fig. 3. Example Spreadsheet that calculates the Cardiac Output

Cell B9 is faulty: it multiplies cells B4 and B6, and should be multiplying cells B4 and B5. The value in cell B10, one of the output cells, is not as expected. The following model was built to represent this problem:

```
**CONSTRAINTS**
watched-or({element(ab,0,1), difference(b4,b6,b9)})
watched-or({element(ab,1,1), product(b9,b6,b10)})
#TEST CASE
eq(b4,120)
eq(b5,50)
eq(b6,72)
```

```
eq(b9, 70)
eq(b10,5040)
```

The model submitted to MINION properly identifying the faulty cell (B9, first variable of the array).

Finally, we have used a spreadsheet from the EUSES Repository to test ConBug concept in a real end-user made spreadsheet (see Figure 4).

	A	B	C	D	E
1	**FINANCIAL SUMMARY FORM (FOR GOS I AND II APPLICANTS ONLY)**				
2					
3	Fill out the summary information below:				
4					
5	The cost of standard office equipment, including computers, may be included in your budget.				
6					
7	Do not include capital expenses for equipment. This equipment would include items that your organization owns or intends to purchase that have a life expectancy of more than three years and a monetary value of more than $500.				
8					
9	Do not include expenses related to the renovation or new construction of cultural facilities				
10					
11	Do not include major purchases related to the renovation or new construction of cultural facilities.				
12					
13	Applicant Name:				
14					
15	Most recently completed fiscal year ended: (month/day/year):				
16					
17		Actual 2002	Actual 2003	Approved 2004 Budget	Projected 2005 Budget
18	1. Earned Income	1,000	1,500	800	1,300
19	2. Contributed Income	500	600	700	800
20	3. Total Income				
21	(line plus 2)	1,500	2,100	2,200	2,100
22	4. Operating Expenses	400	500	600	700
23	5. Net Income/Lose				
24	(line 3 minus line 4)	1,100	1,600	1,600	1,400
25					
26	If you organization is projecting an increase or decrease of 15% or more from one year to the next, explain why:				
27					
28	If you show a surplus, indicate what it will be used for:				
29					
30	If you show a deficit, describe your deficit reduction plan:				

Fig. 4. Example Spreadsheet from EUSES Repository

The spreadsheet was edited to contain a fault on the formula of the cell D20. This leads to an unexpected output on cell D23. Because the constraint list of this spreadsheet model is too long, it is not presented here. After executing the model in MINION, one obtains a diagnosis candidate which identifies correctly the faulty cell (D20, third variable of the array).

On the reported case studies the results were rather precise: our approach managed to properly find the root cause of the observed failure. We now present the performance regarding the run-time of our tests:

- Inversors Problem: 6 constraints in 0.15 seconds
- Example from [17]: 9 constraints in 0.16 seconds
- Cardiac Output Spreadsheet: 7 constraints in 0.17 seconds
- Spreadsheet from EUSES: 32 constraints in 0.17 seconds

These results show that ConBug can be used to debug faulty (real) spreadsheets. The results were obtained using a Sony Vaio VGN-FW51ZF laptop, using Ubuntu Linux 10.10 (64 bit version) and MINION 0.12.

6 Related Work

The work presented in this paper is based on model-based diagnosis [18], namely its application to (semi-)automatic debugging (e.g., [19]). In contrast to previous

work, the work presented of this paper, however, 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 recently in [20] to aid debuggers in pinpointing software failures.

The WHYLINE system, implemented in the Alice environment, allows users to ask questions about expected program behavior [21]. The system uses static and dynamic analyses of the program to help the user locate the cause of the error. Empirical evaluations reported in the paper have shown that users debug errors up to 8 times as fast with WHYLINE than without.

GoalDebug [22] 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 formulas that, when applied, would result in the user-specified output. In [22] a thorough evaluation of the tool is stated. GoalDebug employs a similar constraint-based approach as the one presented in this paper. Moreover, it also suggests a list of changes to fix the speradsheet (which is not currently supported by ConBug).

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 [23]. When a user indicates one or more program failures during this testing process, fault localization techniques [24] 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 formulas. 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 formulas, e.g., [25, 26]. The tools differ in the rules they employ and also 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 [27], which by exploiting techniques for automated header inference [25], can perform unit analysis fully automatically. 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 spreadsheet include code inspection [28] and adoption of better spreadsheet design practices [29]. However, none of these approaches focus on debugging of spreadsheets.

7 Conclusions and Future Work

In this paper, ConBug, a a constraint-based approach for debugging (automatically) spreadsheets was proposed. 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 (an example of a diagnosis candidate is cell B1 and cell C4 are faulty, i.e., explain the faulty

observed value in, e.g., cell A100). We have used three small spreadsheets plus a somewhat more realistic spreadsheet taken from the large EUSES Spreadsheet Corpus to show that the approach 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 [30] 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 [19], which is amongst the best approaches for software fault localization. Fourth, currently our approach only deals with integer cells, we plan to extend our approach to be able to handle, e.g., strings and floats. Finally, we also plan (that is actually on-going work) to evaluate the current approach using a larger set of spreadsheets.

References

1. 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 Computing Surveys (2011)
2. Chadwick, D., Knight, B., Rajalingham, K.: Quality control in spreadsheets: A visual approach using color codings to reduce errors in formulae. Software Quality Journal **9**(2) (2001) 133–143
3. Tukiainen, M.: Uncovering effects of programming paradigms: Errors in two spreadsheet systems. In: Proc. PPIG'00. (2000) 247–266
4. Collavizza, H., Rueher, M.: Exploring different constraint-based modelings for program verification. In: Proc. CP'07, Berlin, Heidelberg, Springer-Verlag (2007) 49–63
5. Ceballos, R., Gasca, R.M., Borrego, D.: Constraint satisfaction techniques for diagnosing errors in design by contract software. ACM SIGSOFT Software Engineering Notes **31**(2) (2006)
6. Wotawa, F., Nica, M.: On the compilation of programs into their equivalent constraint representation. Informatica Journal **32** (2008) 359–371
7. Woods, S., Yang, Q.: Program understanding as constraint satisfaction: Representation and reasoning techniques. Automated Software Eng. **5** (April 1998) 147–181
8. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: ISSTA'98. (1998) 53–62
9. Gotlieb, A., Botella, B., Rueher, M.: A clp framework for computing structural test data. In: Proc. CL'00, London, UK, Springer-Verlag (2000) 399–413
10. Peischl, B., Wotawa, F.: Automated source-level error localization in hardware designs. IEEE Des. Test **23** (January 2006) 8–19
11. Abreu, R., Mayer, W., Stumptner, M., van Gemund, A.J.C.: Refining spectrum-based fault localization rankings. In Wainwright, R.L., Haddad, H., eds.: Proc. SAC'09, Honolulu, Hawaii, USA, ACM Press (8 – 12 March 2009) 409–414

12. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In McMinn, P., ed.: Proc. TAIC PART'07, Windsor, United Kingdom, IEEE Computer Society (September 2007) 89–98
13. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proc. ICSE'09, Washington, DC, USA, IEEE Computer Society (2009) 364–374
14. Mayer, W.: Static and Hybrid Analysis in Model-based Debugging. PhD thesis, School of Computer and Information Science, University of South Australia (2007)
15. Nica, M., Weber, J., Wotawa, F.: How to debug sequential code by means of constraint representation. In Grastien, A., Stumptner, M., eds.: Proc. DX'08, Blue Mountains, NSW, Australia (September 2008) 7–14
16. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast, scalable, constraint solver. In: Proc. ECAI'06, Amsterdam, The Netherlands, The Netherlands, IOS Press (2006) 98–102
17. Nica, M., Nica, S., Wotawa, F.: Does testing help to reduce the number of potentially faulty statements in debugging? In: Proc. TAIC PART'10. (2010) 88–103
18. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1) (April 1987) 57–95
19. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In Taentzer, G., Heimdahl, M., eds.: Proc. ASE'09, Auckland, New Zealand, IEEE Computer Society (16 – 20 November 2009)
20. 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: Proc. CAEPIA'09. (2009) 22–31
21. Ko, A.J., Myers, B.A.: Designing the whyline: a debugging interface for asking questions about program behavior. In: Proc. CHI'04, New York, NY, USA, ACM (2004) 151–158
22. Abraham, R., Erwig, M.: Goaldebug: A spreadsheet debugger for end users. In: Proc. ICSE'07. (2007) 251–260
23. 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: Proc. ICSE'00, New York, NY, USA, ACM (2000) 230–239
24. Ruthruff, J., Creswick, E., Burnett, M., Cook, C., Prabhakararao, S., Fisher, II, M., Main, M.: End-user software visualizations for fault localization. In: Proc. SoftVis'03, New York, NY, USA, ACM (2003) 123–132
25. Abraham, R., Erwig, M.: Header and unit inference for spreadsheets through spatial analyses. In: Proc. VLHCC'04, Washington, DC, USA, IEEE Computer Society (2004) 165–172
26. Ahmad, Y., Antoniu, T., Goldwater, S., Krishnamurthi, S.: A type system for statically detecting spreadsheet errors. Volume 0., Los Alamitos, CA, USA, IEEE Computer Society (2003) 174
27. Abraham, R., Erwig, M.: Ucheck: A spreadsheet type checker for end users. J. Vis. Lang. Comput. **18** (February 2007) 71–95
28. Panko, R.R.: Applying code inspection to spreadsheet testing. J. Manage. Inf. Syst. **16** (September 1999) 159–176
29. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: Proc. VL/HCC'10. (2010) 93–100
30. Abraham, R., Erwig, M.: Mutation operators for spreadsheets. IEEE TSE **35**(1) (2009) 94–108