

Goal-Directed Debugging of Spreadsheets^{*}

Robin Abraham and Martin Erwig
School of EECS, Oregon State University
[abraharo|erwig]@eecs.oregonstate.edu

Abstract

We present a semi-automatic debugger for spreadsheet systems that is specifically targeted at end-user programmers. Users can report expected values for cells that yield incorrect results. The system then generates change suggestions that could correct the error. Users can interactively explore, apply, refine, or reject these change suggestions. The computation of change suggestions is based on a formal inference system that propagates expected values backwards across formulas. The system is fully integrated into Microsoft Excel and can be used to automatically detect and correct various kinds of errors in spreadsheets. Test results show that the system works accurately and reliably.

Keywords: Spreadsheet, Debugging, Static Analysis, End-User Software Engineering.

1 Introduction

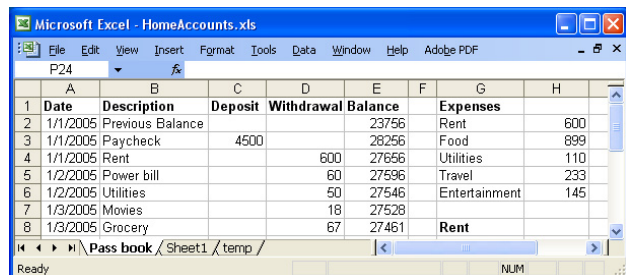
The widespread use of spreadsheets by end-user programmers has led to a situation where up to 90% of spreadsheets have non-trivial errors in them [12]. This has generated considerable interest in the end-user-programming research community to develop tools and systems that would help end users develop correct spreadsheets. The approaches can be principally grouped into two categories:

- (1) Detection of errors [13, 6, 2] and
- (2) Error prevention [5]

The error-detection tools that are currently available are limited in that they only do part of the work—they help identifying the errors, but they do not really help in removing them. Most tools try to mitigate this problem by giving additional fault localization information [14] and better error messages [11] to help the users correct the spreadsheets. However, it would be even more helpful if the tools could suggest and perform corrective changes that would rectify the error in the spreadsheet.

Consider the spreadsheet shown in Figure 1, which shows the monthly income versus expenditure figures. Column B has the descriptions for the different transactions on the user's bank account. Column C has the figures for the

various deposits made to the account, and column D has the figures for the withdrawals from the account. Column E shows the current running balance that is updated after each transaction. The user also likes to keep track of a summary of monthly expenditure. The summary categories are in column G, and the corresponding values are in column H. Finally, G8 has a formula that identifies the category under which the maximum expense has occurred for the month.



	A	B	C	D	E	F	G	H
1	Date	Description	Deposit	Withdrawal	Balance	Expenses	Expenses	
2	1/1/2005	Previous Balance			23756	Rent		600
3	1/1/2005	Paycheck	4500		28256	Food		899
4	1/1/2005	Rent		600	27656	Utilities		110
5	1/2/2005	Power bill		60	27596	Travel		233
6	1/2/2005	Utilities		50	27546	Entertainment		145
7	1/3/2005	Movies		18	27528			
8	1/3/2005	Grocery		67	27461	Rent		

Figure 1. Monthly Expenditure Sheet

At the end of January, the user notices two problems with this spreadsheet. First, the summary figures for food-related expenses is very high compared to previous months—the user suspects the amount \$899 in the spreadsheet is incorrect since it is usually less than \$500. Second, G8 shows Rent as the category under which the maximum expense has occurred whereas it is obvious from the computed values that food-related expenses (expected output Food) have depleted the user's bank account the most. To debug this spreadsheet, the user would have to go over the formulas in cells G8 and H3 carefully to identify the error. If the problem is still unclear, the user would have to identify and go over all the cells referenced by the formula in cell H3. In a more complicated spreadsheet, this problem could be compounded further if the cells referenced by H3 themselves have formulas in them. This makes debugging a challenging and potentially error-prone activity.

In the approach presented in this paper, the user can specify the expected value of an erroneous cell. In case the user does not know the exact output value, she can simply specify a range. For example, in the case discussed above, the user might not know the correct output value for H3, and could specify that some value less than 500 is the expected output from the cell. The system uses this information to infer possible value or formula changes to cells that impact

^{*}This work is partially supported by the National Science Foundation under the grant ITR-0325273 and by the EUSES Consortium (<http://EUSESconsortium.org>).

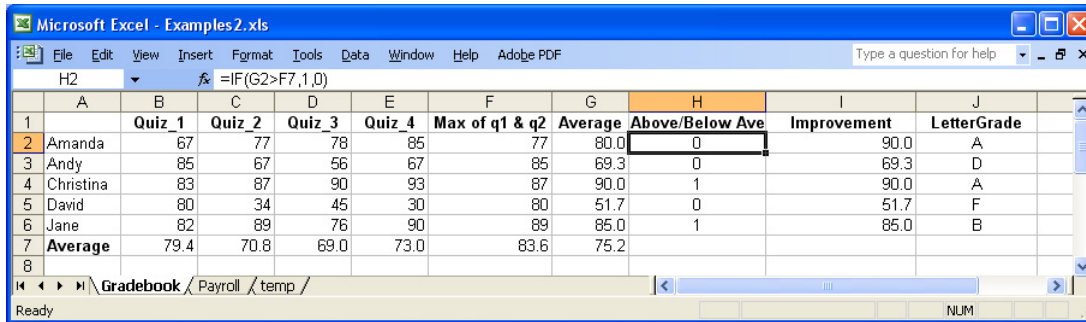


Figure 2. Gradebook spreadsheet

the value in the error cell. In the first step, only the cells that directly influence the value of the erroneous cell are considered. The system uses a set of heuristics to rank the change suggestions since they are not all equally likely. For each of the cells under consideration, five suggestions with the highest ranks are then presented to the user. The user can then proceed in different ways.

- (1) The user might choose to *apply* any one of the presented suggestions. This action would result in the spreadsheet being modified according to the suggestion.
- (2) The user might ask for *more* suggestions. In this case, the constraints are propagated backwards through the sheet where possible, and more suggestions will be generated. All the suggestions will be ranked again on the basis of the heuristics and the higher ranked ones are presented to the user. If the constraints cannot be propagated backwards, the system informs the user that no more suggestions can be generated.
- (3) The user might *reject* one or more of the presented suggestions. The system employs these rejected suggestions as additional constraints in the further generation of suggestions.

In the next section, we discuss how our system could help users debug their spreadsheets. Section 3 presents formal semantics of change inference and the heuristics we have used to rank the suggestions generated by the system. In Section 4, we describe in detail how the system generates change suggestions. Some preliminary steps we have taken to evaluate our approach are described in Section 5. We present related work in Section 6. Conclusions and directions for future work are given in Section 7.

2 Beyond Debugging

The typical scenario of existing debuggers involves finding out how input values have been used to cause a wrong result to be computed. In our system, we invert the “debugging question”: We determine how to change the input cells or formulas so that the correct result will be produced. We

adopt an approach similar to the one described in [8, 15] where the system allows the user to mark a bug and specify expected behavior of the system. In the Whyline system, the users can ask “Why did...?” and “Why didn’t...?” questions to inspect the runtime behavior of the program. Whyline answers the “Why didn’t...?” questions by considering all possible runtime actions that did and did not happen. In our system, users provide constraints describing the expected output values for the cells that are incorrect. This extra information regarding the expected output is then used as described formally in Section 3 to generate change suggestions that are presented to the user.

Figure 2 shows a spreadsheet used to store the grades of the students in a course.¹

Column H checks to see if a student is “above average” (designated by 1) or “below average” (designated by 0) compared to the class average for the course, computed in cell G7. The user might notice that Amanda’s average score is 80, which is above the class average of 75.2. Therefore, the value in H2 should be 1 instead of 0. The value is incorrect because the formula in H2 incorrectly compares G2 with F7 instead of G7. The user can specify the expected value for H2 in the change-request mechanism (shown in Figure 3) that can be invoked by right-clicking in H2 and selecting Debug from the menu.

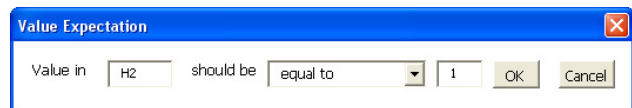


Figure 3. Specifying expected value in a cell

The system generates change suggestions based on the input from the user. The generated suggestions are ranked on the basis of the heuristics we will discuss in Section 3.2. The cell with the highest ranked change suggestion is shaded orange, and all other cells for which change suggestions have been generated are shaded yellow. In the cur-

¹A similar Forms/3 spreadsheet with grade information for one student was used in empirical studies aimed at testing effectiveness of fault localization techniques [14].

rent example, suggestions are generated only for H2 in the first step. The user can view the change suggestions by right-clicking on the cell and selecting Suggestions from the menu. The user can select or ignore change suggestions from the list. We see in Figure 4 that the suggestion for changing F7 to G7 is the second item on the list of possible changes (of which only the five highest ranked are shown) that would result in the formula in H2 evaluating to 1.

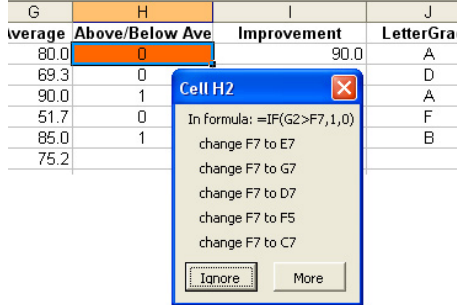


Figure 4. List of suggestions for H2

This change would correct the error in the spreadsheet. If the user picks this change suggestion, she is presented with the window shown in Figure 5 which allows her to make modifications to the suggested formula (if necessary) before applying it to the cell. Once the user clicks Apply the chosen change suggestion is performed and the spreadsheet is error free. We see in this example that only 3 actions/steps were performed by the user (specifying the expected value for the cell with the error, selecting the change suggestion, and applying the change suggestion) in correcting the error. The cognitively difficult task of coming up with the correct suggestion and entering it as the cell’s formula (ensuring syntactic correctness) was taken care of by the system.

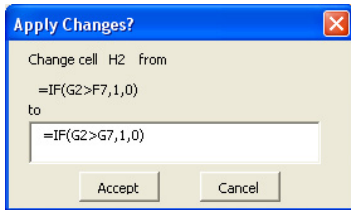


Figure 5. Applying the change suggestion

3 Change Inference

The basic idea of change inference is fairly simple: Given a value w , which represents the expected value for a cell a , determine possible changes in the cell’s formula and in formulas of referenced cells that would cause the actual value computed for the cell a to be w . In fact, we can consider the more general problem of specifying a constraint on the result value of a cell. Initially, these constraints can be of the form ωw where $\omega \in \{<, \leq, =, \geq, >\}$. Propagation

of constraints across cell formulas requires to also include *and* and *or* constraints. A particular challenge for change inference is that, in general, many possible changes exist and that we want to report to the user only a limited set of the most promising or most likely changes.

For the following discussion, we regard a spreadsheet (s) as a mapping from addresses to formulas. The formula stored at the address a in s is obtained by $s(a)$. The evaluation of a formula f to a value v in the context of a spreadsheet s is written as $f \xrightarrow{s} v$. For a spreadsheet s we are given a *target cell* with address a and a *target constraint* γ as defined in Figure 6. Lambda abstractions are needed to define constraint transformations.

$$\begin{aligned} \gamma &::= \omega v \mid \gamma \wedge \gamma \mid \gamma \vee \gamma \mid \lambda x. \gamma \\ \omega &::= < \mid \leq \mid = \mid \geq \mid > \end{aligned}$$

Figure 6. Syntax of constraints.

Constraints effectively describe predicates on values, so that the application of a constraint γ to a value v , written $\gamma(v)$, yields either true or false. For example, $[< 3 \wedge \geq 1](2)$ yields true. We assume that the formula f that is stored in a evaluates to a value v for which $\gamma(v)$ yields false, because otherwise the cell computes a correct value, and no debugging is needed.

We perform change inference in several steps: First, we determine possible changes for a given target value. This process is described in Section 3.1. After that we rank the results by a heuristic to distinguish more likely changes from less likely ones. Possible heuristics are described in Section 3.2. Changes are suggested in chunks of decreasing relevance according to the computed ranking.

3.1 Change Candidates

Change inference is performed by a function δ , which is formally defined in Figure 7. The generated change suggestions are of the form $\{a : f \rightsquigarrow \gamma\}$ to express that the (sub)formula f that is contained in the cell with the address a should be changed to a value v for which $\gamma(v)$ is true.

To illustrate the definition, let us consider a few example cases. The simplest case is that a cell with address a contains a constant, say v , but the target constraint is γ (with $\neg\gamma(v)$). In this case the only possible change is to change the constant v to a value w with $\gamma(w)$ (as shown in definition (1) in Figure 7).² We will discuss later how a constraint γ is converted into a value in the user interface.

If the cell a contains a formula, say $f(e_1, \dots, e_k)$, which evaluates to v , we have two possibilities to derive a change: Either we can change the formula itself, or we can try to “backpropagate” the target constraint γ to the different arguments e_i . This approach depends on the operation f .

²To be precise, there are actually infinitely many possibilities, for example, changing v to formulas, such as $w + 1 - 1$ or $v + w - v$, but the change to the constant w is obviously the simplest change among those. In what follows we consider only fully evaluated suggestions.

$$\begin{aligned}
\delta(a, v, \gamma) &= \{a : v \rightsquigarrow \gamma\} & (1) \\
\delta(a, f(e_1, \dots, e_k), \gamma) &= \bigcup_{i=1}^k \delta(a, e_i, f^i(e^i)(\gamma)) \cup \{a : f(e_1, \dots, e_k) \rightsquigarrow \gamma\} & (2) \\
\delta(a, \uparrow a', \gamma) &= \delta(a', s(a'), \gamma) \cup \{a : \uparrow a' \rightsquigarrow \uparrow a' \mid s(a') \xrightarrow{s} v \wedge \gamma(v)\} \cup \{a : f(e_1, \dots, e_k) \rightsquigarrow \gamma\} & (3) \\
\delta(a, \text{if } p \text{ then } e \text{ else } e', \gamma) &= \begin{cases} \delta(a, e, \gamma) & \text{if } p \xrightarrow{s} T \wedge e' \xrightarrow{s} v \wedge \neg \gamma(v) \\ \delta(a, e', \gamma) & \text{if } p \xrightarrow{s} F \wedge e \xrightarrow{s} v \wedge \neg \gamma(v) \\ \delta(a, p, = F) \cup \delta(a, e, \gamma) & \text{if } p \xrightarrow{s} T \wedge e' \xrightarrow{s} v \wedge \gamma(v) \\ \delta(a, p, = T) \cup \delta(a, e', \gamma) & \text{if } p \xrightarrow{s} F \wedge e \xrightarrow{s} v \wedge \gamma(v) \end{cases} & \begin{aligned} (4a) \\ (4b) \\ (4c) \\ (4d) \end{aligned}
\end{aligned}$$

Figure 7. Change Inference

In general, we need k constraint transformations f^1, \dots, f^k that can compute the change required for any argument that causes the formula $f(e_1, \dots, e_k)$ to evaluate to a value that satisfies γ . We write $f^i(e_i, \dots, e_{i-1}, e_{i+1}, \dots, e_k)(\gamma)$ to refer to the constraint for the i^{th} argument of f . This constraint is always defined such that

$$f^i(e_i, \dots, e_{i-1}, e_{i+1}, \dots, e_k)(\gamma) = \gamma' \implies (\forall v. \gamma'(v) \implies \gamma(f(e_i, \dots, e_{i-1}, v, e_{i+1}, \dots, e_k)))$$

For example, for $+$ the two constraint transformations are defined as follows.

$$\begin{aligned}
+^1(v_2)(\gamma) &= \lambda x. \gamma(x - v_2) \\
+^2(v_1)(\gamma) &= \lambda x. \gamma(x - v_1)
\end{aligned}$$

Now if a cell contains the formula $3 + 5$ but should evaluate to a value that satisfies the constraint > 11 , $+^1$ tells to change 3 to $\lambda x. [> 11](x - 5) = \lambda x. [> 6](x) = (> 6)$ whereas $+^2$ asks for the change of 5 to $\lambda x. [> 11](x - 3) = \lambda x. [> 8](x) = (> 8)$. Both constraints can be converted by the function \mathcal{V} (see below) into values (here, integer values 7 and 9, respectively) so that with either one of the inferred changes we obtain formulas that correctly compute a result that is larger than 11, that is, $7 + 5$ or $3 + 9$. In general, we obtain k suggested changes for a function of k arguments, which leads to the formula employed in definition (2) of Figure 7. In that formula the notation e^i represents the sequence $e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_k$.

The third case to consider is when a formula is a reference to another cell. In that case, the change is inferred for the referenced cell and its content. We can also change the reference to any other cell a'' that evaluates to a value that satisfies γ , or we can replace the reference by the constant itself. This case is covered by definition (3) in Figure 7.

Finally, we distinguish four cases for a conditional formula depending on (i) whether or not the predicate is true and (ii) whether or not one of the alternatives evaluates to a value satisfying γ . For example, consider that the cell a contains the formula $f = \text{if } p \text{ then } e \text{ else } e'$. If the condition p evaluates to true, f evaluates to its first alternative, that is, $f \xrightarrow{s} v$ where $e \xrightarrow{s} v$ with $\neg \gamma(v)$. Therefore, reasonable change suggestions can be obtained through $\delta(a, e, \gamma)$. Should in addition e' evaluate to w with $\gamma(w)$, any change that causes p to evaluate to false is also a reasonable change.

The two other cases are obtained by an analogous consideration of p evaluating to false and e evaluating to w with $\gamma(w)$. The four cases are summarized in definition (4) of Figure 7.

We can observe that the function δ propagates *constraints* through formulas while the system reports *values* in the user interface. The change constraints that are derived by δ will be converted, if possible, into values by a function \mathcal{V} . Before \mathcal{V} is applied, the constraint to be converted is simplified as much as possible. For example, $< 3 \wedge \leq 1$ can be simplified to ≤ 1 . After that \mathcal{V} can produce value suggestions for constraints that do not contain \wedge or \vee .

$$\begin{aligned}
\mathcal{V}(\omega v) &= v \text{ for } \omega \in \{\leq, =, \geq\} \\
\mathcal{V}(< v) &= \max_T \{w \mid w < v\} \\
\mathcal{V}(> v) &= \min_T \{w \mid w > v\} \\
\mathcal{V}(\gamma) &= \gamma
\end{aligned}$$

Note that \max_T and \min_T are type-dependent maximum and minimum functions. For example, $\max_T \{w \mid w < 3\}$ yields 2 if w is an integer, while it yields 2.99 if w is a floating point value.³ In cases when γ is a non-simple constraint, the user is presented with a suggestion that is a textual description of the constraint itself. For example, the suggestion $\{a : 7 \rightsquigarrow \geq 1 \wedge \leq 3\}$ is presented as ‘‘Change 3 to a value between 1 and 3’’. In general, these verbal descriptions are quite difficult to read. An explanation component could be added that tries to help the user come up with suitable values, but currently it is not clear how frequently non-simple constraints really occur.

3.2 Heuristics for Selecting and Presenting Change Suggestions

In general, the system generates many change suggestions. We use a set of heuristics to rank the generated suggestions from unlikely (ranked 0) to most likely (ranked 10). We then use the cell-shading mechanism to direct the user’s attention to the cell(s) with the highest ranked suggestion(s). The heuristics used to rank the change suggestions are discussed in the following.

Formula changes. For ranking the suggestions for formula changes, we use the idea of node equivalence classes

³The number of decimal places is currently fixed to 2. This should be a user-definable parameter.

for formulas from [9]. A partial order on equivalence classes can be defined based on a similarity measure between the original formula in the cell and the suggested change. The greater the similarity, the higher the rank of the corresponding change suggestion.

For example, two formulas are considered to be *copy equivalent* if they are identical when the relative references are compared in the R1C1 notation. Two formulas are considered to be *structurally equivalent* if they contain the same operations in the same order. Copy equivalence implies that the original formula and the suggested change are more similar than would be the case if they were only structurally equivalent. Therefore, if a change suggestion recommends changing a formula f to another one g that is copy-equivalent to f , the change suggestion is ranked high. On the other hand, if a change suggestion recommends changing a formula f to another one g that is only structurally equivalent to f , this suggestion would be ranked lower. Even lower ranked are changes to structurally non-equivalent formulas. In our system, this can only happen in form of value changes, that is, a formula is changed to a value. For example, a change A2+2 to 5 would only receive a very low ranking.

Reference changes. Since our primary focus is on user-generated quantitative errors, we rank the change suggestions that deal with changes to references based on their Manhattan distance from the original reference. In the example shown in Figure 2, the erroneous reference in the formula in H2 points to F7. The suggestion for changing the reference to G7 or E7 receives a higher rating than the suggestion for changing it to, say D7, since G7 and E7 are only one cell away from F7 whereas D7 is 2 cells away. This heuristic is based on the assumption that the introduction of the incorrect reference is primarily the result of a mechanical error (clicking an incorrect cell to select the target). This makes closer cells more likely suggestions.

Value changes. Generating specific values in change suggestions is hard in some cases since the system might not have enough constraints to do so. Therefore, value suggestions generally specify acceptable ranges. We rank value suggestions on the basis of their types. For example, a change suggestion that recommends changing an integer value to a float would have a lower rank than a suggestion that recommends changing an integer value to some other integer value. A naive way to get the expected target value in a cell is to replace the formula within the cell with the target value. Even though such suggestions are generated by the system, they are given very low rank.

In addition to the heuristics discussed above, the system also performs some additional checks to ensure that the generated suggestions are reasonable and correct. For example, performing any of the generated suggestions would not introduce a circular reference in the spreadsheet.

4 Change Inference Example

In this section we take a closer look at how the change suggestions are generated by the system. In the Gradebook example shown in Figure 2, the user specifies that the expected value in cell H2 is 1. This information is stored in the system as a constraint on the value in the cell. The formula in the cell is $\text{IF}(\text{G2}>\text{F7},1,0)$. From the semantics of change inference shown in Figure 7, we see that rule (4) needs to be applied. Since the condition $\text{G2}>\text{F7}$ evaluates to false, we might consider applying rule (4b) or (4d) for generating change suggestions. But we see that e in this case is 1, which is the expected value. Therefore, we apply rule (4d), which produces the union of the change suggestions generated by the recursive calls $\delta(\text{H2}, \text{G2}>\text{F7}, = T)$ (changes that make the predicate evaluate to true), and $\delta(\text{H2}, 0, = 1)$ (changes that make the false branch of the function evaluate to the expected value). For $\delta(\text{H2}, \text{G2}>\text{F7}, = T)$, rule (2), that is $\delta(a, f(e_1, \dots, e_k), \gamma)$ can be used, where f is the logical operator $>$. The set of generated change suggestions is the union of the following sets.

- (1) $\delta(\text{H2}, \uparrow\text{F7}, < 80)$: Changes to the reference F7 that would give some value less than 80 as the result.
- (2) $\delta(\text{H2}, \uparrow\text{G2}, > 83.6)$: Changes to the reference G2 that would give some value greater than 83.6 as the result.
- (3) $\{\text{H2} : \text{G2} > \text{F7} \rightsquigarrow = T\}$: Effectively replacing the predicate with T (see function \mathcal{V}). This suggestion is ranked very low.

$\delta(\text{H2}, \uparrow\text{F7}, < 80)$ invokes rule (3) and results in the following suggestions.

- (4) $\delta(\text{F7}, \text{AVERAGE}(\text{F2}:\text{F6}), < 80)$: This change suggestion recommends that the formula in F7 should be changed so that the computed value in the cell changes from 83.6 to some value less than 80. In the first step, this is simply stored as a system-generated constraint and does not generate any concrete change suggestions. When the user rejects the suggestions generated at the first level, or explicitly asks for more suggestions, the propagated constraints are used to populate the change-suggestion queue.
- (5) $\{\text{H2} : \uparrow\text{F7} \rightsquigarrow \uparrow a' \mid s(a') \xrightarrow{s} < 80\}$: These suggestions give rise to all the addresses of the cells that have values less than 80 (for example, E7, G7, D7, C7, etc.).
- (6) $\delta(\text{H2}, \uparrow\text{F7}, < 80)$: This change suggestion recommends replacing the reference to cell F7 with some value less than 80.

The case for $\delta(\text{H2}, \uparrow\text{G2}, > 83.6)$ is similar to the case discussed above.

Once the change suggestions have been generated, the system ranks them on the basis of the heuristics described in Section 3.2. The suggestions are then presented to the users when they ask for it. In the example shown in Figure 4, the

suggestions generated from step 5 above include the change that would remove the error from the sheet, that is replacing reference to F7 in the formula in H2 with reference to G7. Consider a scenario in which the user does not notice this suggestion right away and instead decides that the first suggestion (changing F7 to E7) is incorrect and rejects it. The system would convert this additional information to a new constraint that would filter out all change suggestions that would recommend changing F7 to E7 for the cell H2. The system would also use propagated constraints (for example, the one shown in item 4 above) to generate new change suggestions (so that they can take the place of the suggestions that have been rejected by the user). The complete set of suggestions are then ranked once again according to the heuristics and then presented to the user.

5 Evaluation of the System

During empirical studies to evaluate the effectiveness of fault-localization techniques [14] in the WYSIWYT system [13], it was observed that subjects make many wrong decisions (*oracle mistakes*) while performing their tasks [10]. Oracle mistakes are incorrect decisions made by users during testing. In this section, we show how our system can prevent many of these mistakes.

Two spreadsheets, seeded with errors, were used in the studies. The first one, shown in Figure 2, computes the grades for the students in a course, and the second sheet, shown in Figure 9, computes the payroll figures for an employee. The subjects start with a sheet in which all the input cells have been set to zero. As part of the task description, the subjects are also given two test cases for each spreadsheet, and an informal specification of the spreadsheet that explains how the different output values are to be computed. Two sample test cases for the Payroll sheet are shown in Table 1. The test cases specify sample input values and expected output values. The subjects then have to come up with additional test cases, inspect the output and decide if it is correct or not.⁴ If the output is incorrect, the users can indicate that to the system by placing a \times in the cell. Similarly, correct output values can be indicated by putting a \checkmark in the cell. The WYSIWYT fault localization mechanism then uses the user feedback to shade the cells depending on their fault likelihood.

We now take a look at how the user could go about debugging the Payroll sheet using our system. After entering the input values from the test cases, the user might notice that the value in the cell T2 is incorrect. More precisely, the expected value according to the test case is 5887 while the value currently in the cell is 5587. The user can specify

⁴The Forms/3 system, in which the WYSIWYT approach has been implemented, has an automatic test-case-generation mechanism called “Help me test” (HMT) described in [7]. The studies we are discussing in this section did not involve the use of the HMT system—the users had to come up with the test cases by themselves based on their understanding of the spreadsheet specifications.

Name	Joe	Mary
Marital Status	S	M
Allowances	1	5
Gross pay	6000	8000
Gross pay YTD	54000	72000
Pre-tax child care	0	400
Life insurance policy amount	10000	50000
Health insurance premium	390	480
Dental insurance premium	18	39
Life insurance premium	5	25
Employee insurance cost	413	544
Employer insurance contribution	300	520
Net insurance cost	113	24
Adjusted gross pay	5887	7576
Federal income tax withheld	551.80	607.80
Social security tax	372	496
Medicare tax	87	116
Total employee taxes	1010.80	1219.80
Net pay	4876.20	6356.20

Table 1. Sample test data for Payroll sheet

this to the system by bringing up the debugging interface as described in Section 2. The system generates the change suggestions, and the cell with the highest ranked change suggestion is shaded orange as shown in Figure 8. Notice that for this cell only one suggestion—the correct one—is generated.

Medicare	Life insur. prem.	Health insur. prem.	Dental insur. prem.	Adjust. gross pay	Empl. insur. cost	Empl. insur. contrib.	Net insur. cost	Empl. taxes
87	5	390	18	5587	413	300	113	601.3
116	25	480	39					4
130.5	4	390	18					6

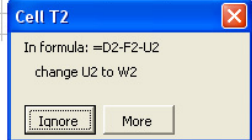


Figure 8. Change suggestion for cell T2

The original (incorrect) formula in T2 was D2-F2-U2. The system recommends that the reference to U2 be replaced with a reference to W2. The part of the spreadsheet specification provided to the user that explains how “Adjusted Gross Pay” is to be computed is shown below. “Pre-tax deductions (such as child care and employee insurance expense above the employer’s insurance contribution) are subtracted from Gross Pay to obtain Adjusted Gross Pay.”

We see that the suggestion generated by the system is correct since the formula should reference the net insurance cost to the employee (computed in W2) and not the total insurance cost before the employer’s contribution has been deducted (computed in U2). If the user applies the suggestion, the formula in T2 is changed to D2-F2-W2, and the computed value in the cell will be 5887 (the expected value according to the test case).

Further comparison of the spreadsheet output with the test case values reveals that the cell L2 has the value 538.9 whereas the expected value is 551.8. When the user pro-

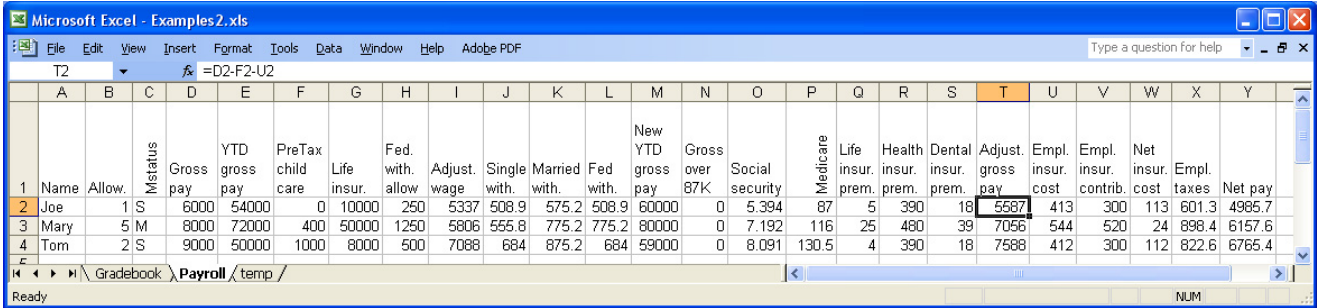


Figure 9. Payroll spreadsheet

vides this information to the system, the suggestions shown in Figure 10 are generated for the cell J2.

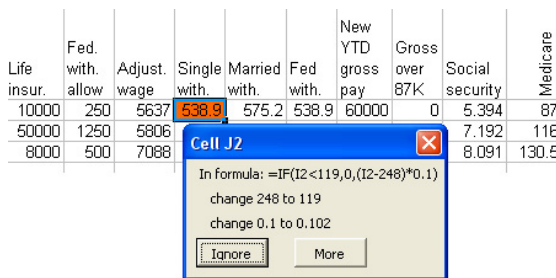


Figure 10. Change suggestion for cell J2

No suggestions are generated for L2 in the first stage since L2 has the formula $IF(C2="S",J2,K2)$, and the value in C2 is "S". The formula in J2 is $IF(I2 < 119, 0, (I2 - 248) * 0.1)$. The highest ranked suggestion recommends that the value 248 be changed to 119. The relevant part of the spreadsheet specification that explains how the federal income tax withholding is computed is shown below.

If single and the adjusted wage is not greater than \$119, the withholding tax is \$0; otherwise the withholding amount is 10% of (adjusted wage - \$119).

Again, the highest ranked suggestion generated by our system is correct since the amount to be deducted from the adjusted wages of a person who is single is \$119 and not \$248 as is in the formula. The user can apply this suggestion, and the cell is evaluated to the expected correct value 551.8.

The numbers for the incorrect testing decisions made by the users of the WYSIWYT system are shown in Table 2. We see that the subjects made 154 oracle mistakes in the Gradebook task and 381 oracle mistakes in the Payroll task. We also see from the data that there were 81 instances during the Gradebook task and 68 instance during the Payroll task when the subjects edited formulas that were actually correct and introduced errors in the spreadsheets. Our system would not have protected the subjects from these mistakes. On the other hand, we also see from the data that there were 373 instances in the Gradebook task and 293 instances in the Payroll task when the subjects pinpointed the

cells with errors correctly but then went on to make incorrect changes to the formulas in the cells. Our system would have prevented all these errors.

	Gradebook	Payroll
Number of subjects	51	51
Total errors	154	381
Errors on values	144	168
Errors on formulas	10	213
Formula-edit errors		
Correct to incorrect	81	68
Incorrect to incorrect	373	293

Table 2. User mistakes during debugging

To summarize, in cases in which the subject incorrectly identified a correct cell as incorrect, our system would not be very helpful. It would simply generate change suggestions that compute the target value specified by the subject. Unfortunately, this problem cannot be avoided or overcome as long as the user has the last word on whether a cell is correct or incorrect. However, for the other cases in which the user correctly identifies an error cell, our system generates the suggestions that correct the error, and performs them accurately, which avoids a whole class of errors. Moreover, for the cases discussed above, the correct change suggestions are ranked highest.

6 Related Work

The main focus of research into reducing the incidence of errors in spreadsheets has been on testing [13] and consistency checking [6, 4, 1]. We have also been working on an approach that avoids formula errors in spreadsheets by generating correct spreadsheets from a predefined specification [5].

The "data validation" tool in Microsoft Excel allows the user to specify the acceptable values for a cell. The system warns the user whenever there is a violation and is useful in keeping track of potential errors in the spreadsheet. It does not really help with debugging per se since Excel does not do any reasoning with the user-specified allowed values. The "trace error" feature in Excel allows the user to incrementally step through the spreadsheet dataflow graph

and inspect predecessors and dependents of cells. This technique tends to be tedious and error prone in large and complex spreadsheets.

In the “interval testing” technique described in [3], users can enter allowed ranges of values for cells. The system uses this information to calculate allowed intervals on cells that are dependent on those for which the ranges have been specified by the user. In case of conflict, heuristics are used to determine the “most influential faulty cell”. The propagation of intervals over some functions (for example, if statements) is not trivial and has not been addressed in [3]. In the approach discussed in [4], the system generates a set of assertions based on assertions entered by the user. The system warns the user when there is a conflict between the system-generated assertions and the user-specified assertions for a cell. The feedback about the conflict in assertions only indicates that there could be an error, either in the user-specified assertions themselves, or in the cell formula. In our system, the information provided by the user about the expected value in a cell can be considered as a user-specified assertion. The system then back-propagates this information as a series of constraints on the values of the cells that contribute to the value in the cell with the error. These constraints are then used to generate change suggestions.

7 Conclusions and Future Work

We have shown that user input of expected values can be exploited to automatically suggest and perform corrective actions to effectively remove errors from spreadsheets. We have also demonstrated that this approach can help to avoid many errors that users make during the debugging process.

The current system has several limitations that we will address in future work. For example, complex constraints cannot be communicated very well to the user. Moreover, the current system does not produce good suggestions for cases in which a cell has an incorrect value due to faults in two or more cells it has references to. In order to keep the number of generated suggestions small, we always propagate constraints only along one argument at a time. One approach to make the system more flexible is to work with more than one initial constraint and then consider different cell orderings for constraint propagation.

Another improvement would be to integrate the system with the UCheck system [1] we have developed based on the work described in [6]. This integration would allow the debugger to generate unit-correct suggestions, thereby adding an additional layer of consistency checking to the debugging process, and further reducing the number of generated suggestions.

Finally, we will look into ways to integrate automatic change suggestions with the WYSIWYT system since testing and debugging are complimentary actions.

References

- [1] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets Through Spatial Analyses. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.
- [2] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A Type System for Statically Detecting Spreadsheet Errors. In *18th IEEE Int. Conf. on Automated Software Engineering*, pages 174–183, 2003.
- [3] Y. Ayalew and R. Mittermeir. Spreadsheet Debugging. In *European Spreadsheet Risks Interest Group*, 2003.
- [4] M. M. Burnett, C. Cook, J. Summet, G. Rothermel, and C. Wallace. End-User Software Engineering with Assertions. In *25th IEEE Int. Conf. on Software Engineering*, pages 93–103, 2003.
- [5] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic Generation and Maintenance of Correct Spreadsheets. In *27th IEEE Int. Conf. on Software Engineering*, pages 136–145, 2005.
- [6] M. Erwig and M. M. Burnett. Adding Apples and Oranges. In *4th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pages 173–191, 2002.
- [7] M. Fisher II, M. Cao, G. Rothermel, C. Cook, and M. M. Burnett. Automated Test Case Generation for Spreadsheets. In *24th IEEE Int. Conf. on Software Engineering*, pages 141–151, 2002.
- [8] A. J. Ko and B. A. Myers. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Conference on Human Factors in Computing Systems*, pages 151–158, 2004.
- [9] R. Mittermeir and M. Clermont. Finding High-Level Structures in Spreadsheet Programs. In *9th Working Conference on Reverse Engineering*, pages 221–232, 2002.
- [10] A. Phalgune, C. Kissinger, M. Burnett, C. Cook, L. Beckwith, and J. Ruthruff. Garbage In, Garbage Out? An Empirical Look at Oracle Mistakes by End-User Programmers. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, 2005. To appear.
- [11] S. Prabhakarao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and Behaviors of End-User Programmers with Interactive Fault Localization. In *IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pages 203–210, 2003.
- [12] K. Rajalingham, D. R. Chadwick, and B. Knight. Classification of Spreadsheet Errors. *Symp. of the European Spreadsheet Risks Interest Group (EuSPRIG)*, 2001.
- [13] G. Rothermel, M. M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A Methodology for Testing Spreadsheets. *ACM Transactions on Software Engineering and Methodology*, pages 110–147, 2001.
- [14] J. Ruthruff, E. Creswick, M. M. Burnett, C. Cook, S. Prabhakarao, M. Fisher II, and M. Main. End-User Software Visualizations for Fault Localization. In *ACM Symp. on Software Visualization*, pages 123–132, 2003.
- [15] B. T. V. Zanden, D. Baker, and J. Jin. An Explanation-Based, Visual Debugger for One-Way Constraints. In *17th Annual ACM Symp. on User Interface Software and Technology*, pages 207–216, 2004.