GoalDebug: A Spreadsheet Debugger for End Users*

Robin Abraham
School of EECS
Oregon State University
abraharo@eecs.oregonstate.edu

Martin Erwig
School of EECS
Oregon State University
erwig@eecs.oregonstate.edu

Abstract

We present a spreadsheet debugger targeted at 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. The change suggestions are ranked using a set of heuristics.

In previous work, we had presented the system as a proof of concept. In this paper, we describe a systematic evaluation of the effectiveness of inferred change suggestions and the employed ranking heuristics. Based on the results of the evaluation we have extended both, the change inference process and the ranking of suggestions. An evaluation of the improved system shows that change inference process and the ranking heuristics have both been substantially improved and that the system performs effectively.

1 Introduction

Programmers spend a major portion of their time debugging code. A recent study conducted in the U.S. by NIST found that software engineers typically spend 70-80% of their time testing and debugging. On average, errors take 17.4 hours to find and fix [30].

The situation is particularly challenging in the area of spreadsheets, which are by far the most widely used programming tools [29]. For example, in the U.S. alone the number of people programming spreadsheets is estimated to be 11 million, compared to only 2.75 million other, professional programmers. Studies have shown that there is a high incidence of errors in end-user spreadsheets, up to 90% in some cases [25]. Some of these errors have resulted in companies losing millions of dollars [14].

Program failures that occur during testing indicate faults that have to be located and corrected during debugging. The WYSIWYT testing methodology [26] has been developed to aid end users in testing spreadsheets. Even with the testing framework, and automatic test-case generation support [3, 15], the *debugging* of spreadsheets is hardly supported. Fault localization techniques [27] seek to help the user with where the program corrections need to be made when faced with one or more program failures during testing. However, there is little help regarding how the program needs to be changed.

Approaches to end-user debugging face two challenges. The first is fault localization: In many cases end users do not know the cause of a particular failure in their spreadsheets. The second difficulty is that, even in cases in which end users are able to identify the cause of a failure correctly, they have trouble with formula syntax and often introduce more errors. To solve both of these problems, a spreadsheet debugger should help the users correctly identify the faults within their programs and eliminate (or at least minimize) the need for the user to apply changes to spreadsheet formulas through manual editing.

The system described in this paper, GoalDebug, addresses these two challenges by automatically generating change suggestions based on the user's expectations about the output of a cell. Moreover, change suggestions can be automatically applied to spreadsheet formulas and obviate the need for formula editing by the end user.

The problem of formula editing is substantiated by data gathered during studies to evaluate the effectiveness of fault-localization techniques [27] in the WYSIWYT system. It was observed that subjects make many wrong decisions (*oracle mistakes*) while performing their tasks [24]. Oracle mistakes are incorrect decisions made by users during testing. The numbers for the incorrect testing decisions found in the study [24] are shown in Table 1. In particular, we can 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. Those errors could not have occurred when using GoalDebug.

A principal difficulty of the GoalDebug approach is caused by the fact that a different output in one cell can, in general, be achieved through many different changes in that or other cells. We employ heuristics to rank possible changes so that the correct one shows up as high as possible in the ranked list of suggested changes to be presented to the user. In the two example spreadsheets discussed above,

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

	Gradebook	**Payroll**
Number of subjects	51	51
Total errors	154	381
Errors on values	144	168
Errors on formulas	10	213
Total formula-edit errors	454	361
Correct to incorrect	81	68
Incorrect to incorrect	373	293

Table 1. User mistakes during debugging

the correct change suggestions are, in fact, ranked highest by GoalDebug.

We have introduced initial ideas of GoalDebug in [2] as a proof of concept. We describe the system in Section 2. To evaluate GoalDebug, we have performed a systematic study of the effectiveness of inferred change suggestions. The idea was to systematically mutate spreadsheet formulas and see whether GoalDebug can suggest highly ranked changes to revert these mutations. This evaluation is described in Section 3. It turns out that certain kinds of mutations could not be inverted at all. Guided by these results, we have extended GoalDebug's change inference, which is described in Section 4.1. The evaluation of this improved version of GoalDebug, described in Section 4.2, showed only partial success, because the employed ranking heuristics were not powerful enough to cope with the significant increase in the number of change suggestions, caused by the extension of change inference. We therefore have added five new components to the ranking heuristics. These are described in Section 5. An evaluation of the final system with extended change inference and improved heuristics is described in Section 5.2 and shows that the new ranking heuristics are significantly better than the old and that the overall effectiveness of the system is good. In Section 6 we discuss related work, and in Section 7 we will present conclusions and discuss directions for future research.

2 Debugging With Suggestions

Given the expected value w for a cell, represented by a constraint, GoalDebug generates possible formula changes that would result in the value w being computed in the cell.

For the following discussion, we regard a spreadsheet (s) as a mapping from addresses (a) to formulas (f). Formulas are either plain values v, references to other cells (given by addresses), or operations (ψ) applied to one or more argument 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 $f \xrightarrow{s}\!\!\!\to v$. To run GoalDebug on a spreadsheet s at a *target cell* with address a, we assume a *target constraint* γ on a, which has the following form (where $\omega \in \{<, \leq, =, \geq, >\}$).

$$\gamma \quad ::= \quad \omega v \mid \gamma \wedge \gamma \mid \gamma \vee \gamma \mid \lambda x.\gamma$$

In addition to value constraints (ωw), *and* constraints allow the formulation of ranges of values as expectations, and *or* (as well as *and*) constraints capture results of constraint propagation. Lambda abstractions are needed to define constraint transformations. Since a constraint γ defines a value predicate, it can be applied to values, as in $\gamma(v)$. For example, $[< 3 \wedge \geq 1](2)$ yields true.

We perform change inference in three steps. First, we find possible changes for a given target value (see Section 2.1). Second, we sort the results by applying a likelihood heuristic (see Section 2.2). Finally, change suggestions are presented to the end user in order of decreasing relevance according to the computed ranking (see [2]).

2.1 Change Inference

Given a value constraint γ, change inference computes a set of change suggestions. Each suggestion is expressed in terms of a constraint γ, which will later be converted into a value. A suggestion has 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. The inference of change suggestions is formalized through the function δ shown in Figure 1.

Let us elucidate the definition with several examples. If a cell with address a contains a constant, say v (*actual value*), but the target constraint is γ (with $\neg\gamma(v)$), the suggestion is to change the constant v to another constant w (*target value*) for which $\gamma(w)$ holds (see def. (1) in Figure 1). How γ can be converted into an actual target value is explained later.

In the case of a formula $f(e_1, \ldots, e_k)$, which evaluates to v, there are basically two possibilities to derive a suggestion: Either change the formula itself, or try to "backpropagate" the target constraint γ to the different arguments e_i, which depends, of course, on the operation f. In general, we need k constraint transformations $f^1, \ldots, f^k$ that can compute the change required for any argument that causes the formula $f(e_1, \ldots, e_k)$ to evaluate to a value that satisfies γ. We abbreviate the sequence $e_1, ..., e_{i-1}, e_{i+1}, ..., e_k$ by e^i and write $f^i(e^i)(\gamma)$ to refer to the constraint for the i^{th} argument of f. This constraint must be defined to satisfy the following implication.

$$\begin{aligned}&f^i(e^i)(\gamma) = \gamma' \Longrightarrow \\ &\qquad (\forall v.\gamma'(v) \Longrightarrow \gamma(f(e_1, ..., e_{i-1}, v, e_{i+1}, ..., e_k))\end{aligned}$$

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

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

To see how this works, consider the case in which a cell contains the formula $3 + 5$ but should evaluate to a value that satisfies the constraint > 11. In this case, $+^1$ derives for the first argument 3 the constraint $\lambda x.[> 11](x - 5)$, which

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

Figure 1. Change Inference

can be simplified to $\lambda x.[> 6](x)$ and further to (> 6). Similarly, $+^2$ derives for 5 the constraint $\lambda x.[> 11](x - 3) = \lambda x.[> 8](x) = (> 8)$. Both constraints can be converted by the function $\mathcal{V}$ (shown below) into values (here, integer values 7 and 9, respectively). Therefore, by applying either suggestion we obtain a formula ($7 + 5$ or $3 + 9$) that correctly computes a result larger than 11. For a function of k arguments we can derive k suggestions, see def. (2).

For a cell reference, changes are inferred for the referenced cell, and the address can be changed to any other cell a'' that evaluates to a value that satisfies γ. Moreover, the reference can be replaced by the constant itself, see def. (3).

Finally, we provide a specialized inference for conditional formulas since we have more detailed information about the data flow from subformulas than in the generic formula case described above. We distinguish four cases depending on the result of the predicate and on whether or not one of the alternatives evaluates to a value satisfying γ. For example, consider the case when the cell a contains the formula $f = \textbf{if}\, p\, \textbf{then}\, e\, \textbf{else}\, e'$. If the condition p evaluates to true, f evaluates to its first alternative, that is, $f \overset{s}{\twoheadrightarrow} v$ where $e \overset{s}{\twoheadrightarrow} v$ with $\neg\gamma(v)$. Therefore, reasonable change suggestions can be obtained through $\delta(a, e, \gamma)$. This case is captured in definition (4a). Should in addition e' evaluate to w with $\gamma(w)$, any change that causes p to evaluate to false is also a reasonable change, see definition (4c). The two other cases, (4b) and (4d), are obtained by an analogous consideration of p evaluating to false and e evaluating to w with $\gamma(w)$.

We can observe that the function δ propagates *constraints* through formulas while the system reports *values* in the user interface. Once δ has propagated the initial constraint into a set of change suggestions, which still contain constraints, these change constraints are converted into values by a function $\mathcal{V}$. First, 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 $\vee$.

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

The functions $\max_T$ 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.[1] In cases when γ is a non-simple constraint that cannot be solved, the user is presented with a suggestion that is a textual description of the constraint itself.

2.2 Suggestion Ranking

Since there can be many suggestions, the ranking heuristics play an important role in minimizing the effort the user has to invest in picking the correct change suggestion to apply. In the following, we describe the ranking heuristics of the original GoalDebug system. Different ranking strategies are applied depending on the *kind* of change suggestion.

For assessing the likelihood of a change to a *formula*, we employ the idea of *node equivalence classes* for formulas [19]. A similarity measure based on these classes indicates the difference of the changed formula from the original one. Since more drastic, that is, less similar changes, seem less likely, higher similarity yields a higher rank for a particular 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 g that is copy-equivalent to f, the change suggestion is ranked higher than a suggestion that would yield a formula h that is only structurally equivalent to f. Even lower ranked are changes to structurally non-equivalent formulas. In GoalDebug this can 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.

Changes to *references* are ranked based on their Manhattan distance from the original reference. This approach makes closer cells more likely suggestions. For example, any suggestion to change B5 in a formula, say 2*B5, to B6 or B4 (or to A5 or C5) would be ranked higher than suggestions for changes to B7, A4, or D5. This heuristic is based

[1] The number of decimal places is arbitrarily fixed to 2.

Operator	Description
ABS	*ABS*olute value insertion
AOR	*A*rithmetic *O*perator *R*eplacement
CRP	*C*onstants *ReP*lacement
CRR	*C*onstants for *R*eference *R*eplacement
LCR	*L*ogical *C*onnector *R*eplacement
ROR	*R*elational *O*perator *R*eplacement
RCR	*R*eference for *C*onstant *R*eplacement
FDL	*F*ormula *DeL*etion
FRC	*F*ormula *R*eplacement with *C*onstant
RFR	*ReF*erence *R*eplacement
UOI	*U*nary *O*perator *I*nsertion
CRS	*C*ontiguous *R*ange *S*hrinking
NRS	*N*on-contiguous *R*ange *S*hrinking
CRE	*C*ontiguous *R*ange *E*xpansion
NRE	*N*on-contiguous *R*ange *E*xpansion
RRR	*R*ange *R*eference *R*eplacement
FFR	*F*ormula *F*unction *R*eplacement

Table 2. Mutation operators for spreadsheets

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) and reflects typical reference errors that occur when users click in a wrong cell while editing formulas.

Finally, *value changes* are ranked based on their types. For example, a change suggestion that recommends changing an integer value to a float (which could be caused, for example, by a division constraint) would have a lower rank than a suggestion that recommends changing an integer value to some other integer value. This ranking does not apply to suggestions to replace formulas by values since these are already covered by the ranking based on similarity, which is very low in this case.

3 Evaluation

The evaluation of GoalDebug has to consider two aspects. First, the system should be able to generate change suggestions to correct errors in the formulas that manifest as program failures. In this context, an *effective change suggestion* is one that corrects the formula error. Second, the system should be able to rank the generated change suggestions so that the correct change suggestions show up high on the list presented to the user. In this context, a more *effective set of ranking heuristics* would rank the correct change suggestions higher than a less effective one. The effectiveness of the generated change suggestions and ranking heuristics are the two parameters we will be measuring the performance of the system on.

To study these two aspects, we systematically mutate formulas and determine whether GoalDebug is able to suggest changes that can revert those mutations and how high they are ranked among all inferred changes.

In previous work, we have developed operators for mutation testing of spreadsheets [5]. These operators were based on mutation operators developed for general-purpose programming languages [20] and on errors committed by end users while creating spreadsheets [8, 23]. One of the goals behind the design of the mutation operators was to automatically seed spreadsheets with errors for empirical studies. The full suite of mutation operators we have developed is shown in Table 2. These operators can be used to model deviations from the correct spreadsheet.

For the evaluation, we used spreadsheets used in empirical studies described in [3,15], see Table 3. For each spreadsheet the following information is given.

1. Number of formula cells in the spreadsheet (Fml).
2. Total number of cells in the spreadsheet (Total).
3. The number of generated *irreversible* mutants. These are formulas that evaluate to the same value as the original formula and thus cannot produce failures that could be identified by the user. GoalDebug is principally inapplicable in those cases and cannot be invoked to generate change suggestions since the computed output and expected output are the same.
4. The number of generated *reversible mutants*. These mutant formulas evaluate to values that are different from the values produced by the original formulas, and GoalDebug can be invoked on those cells.
5. Total number of generated mutants for each sheet.

Sheet	**Cells**		**Mutants**		
	Fml	**Total**	**Irrever.**	**Rever.**	**Total**
Microgen	2	12	143	33	176
GradesNew	8	26	157	181	338
FitMachine	6	18	366	74	440
Digits	6	14	172	293	465
NetPay	6	18	61	47	108
Purchase	15	50	172	153	325
RandJury	21	58	578	308	886
Sales	16	29	0	338	338
Solution	3	12	119	116	235
Budget	6	24	46	112	158
MBTI	28	83	902	243	1145
NewClock	10	24	156	165	321
GradesBig	21	48	283	647	930
Harvest	9	26	10	221	231
Payroll	54	100	347	1057	1404
Total	211	542	3512	3988	7500

Table 3. Sheets used in the evaluation

Obviously, not all operators shown in Table 2 are applicable to all formulas. Therefore, we picked sheets that had many different kinds of formulas to be able to apply the operators from the suite. All operators, except FDL (formula-deletion operator) and FRC (formula replace with constant operator) from the suite were used to seed errors in the spreadsheet. We excluded the FDL operator for two reasons. First, Excel does not handle the inclusion of references to blank cells in spreadsheet formulas consistently.

Sheet	Operators: Uncorrected [Total]								
	AOR	CRP	CRR	LCR	NRE	NRS	RFR	ROR	RRR
Microgen	6 [6]	0 [3]	4 [4]	1 [1]	0 [0]	0 [0]	3 [16]	3 [3]	0 [0]
GradesNew	18 [18]	0 [4]	25 [25]	1 [1]	0 [0]	0 [0]	16 [123]	10 [10]	0 [0]
FitMachine	9 [9]	3 [6]	11[11]	1 [1]	0 [0]	0 [0]	5 [41]	6 [6]	0 [0]
Digits	62 [62]	0 [17]	43 [43]	0 [0]	0 [0]	0 [0]	27 [152]	19 [19]	0 [0]
NetPay	3 [3]	0 [6]	12 [12]	0 [0]	0 [0]	0 [0]	0 [22]	4 [4]	0 [0]
Purchase	14 [14]	0 [4]	29 [29]	0 [0]	0 [0]	0 [0]	3 [91]	15 [15]	0 [0]
RandJury	87 [87]	0 [47]	33 [33]	6 [6]	0 [0]	0 [0]	6 [119]	16 [16]	0 [0]
Sales	72 [72]	0 [12]	49 [49]	0 [0]	0 [0]	0 [0]	24 [205]	0 [0]	0 [0]
Solution	21 [21]	0 [2]	12 [12]	0 [0]	0 [0]	0 [0]	11 [77]	4 [4]	0 [0]
Budget	15 [15]	0 [1]	18 [18]	0 [0]	0 [0]	0 [0]	10 [75]	3 [3]	0 [0]
MBTI	43 [43]	0 [24]	35 [35]	16 [16]	0 [0]	0 [0]	3 [120]	5 [5]	0 [0]
NewClock	20 [20]	1 [11]	22 [22]	1 [1]	0 [0]	0 [0]	0 [97]	14 [14]	0 [0]
GradesBig	6 [6]	3 [5]	28 [28]	1 [1]	99 [99]	27 [27]	12 [103]	14 [14]	272 [364]
Harvest	0 [0]	0 [0]	5 [5]	0 [0]	40 [40]	18 [18]	0 [24]	0 [0]	99 [134]
Payroll	170 [170]	0 [42]	166 [166]	0 [0]	0 [0]	0 [0]	39 [641]	38 [38]	0 [0]
Total	546 [546]	7 [184]	492 [492]	27 [27]	139 [139]	45 [45]	159 [1906]	151 [151]	371 [498]

Table 4. Original version of GoalDebug's effectiveness at correcting mutations

For example, a (reference to a) blank cell in a SUM aggregation is treated as a 0, whereas a blank cell in an IF statement is considered to be smaller than any string or number. A blank cell in a binary operation is flagged as an error. This aspect makes it difficult to model Excel's handling of blank cells. Second, it seems unlikely that the user would mark an empty cell as incorrect and specify the expected output. The FRC operator was excluded because the mutation is not a minor change to reverse. That is, given only the expected output and the actual data in the cell, it is not easy in general to generate a formula that would result in the expected output. Even so, if either of these situation arose, GoalDebug could potentially do one of two things. The first option would be to directly recommend that the specified expected value be entered in the cell. The second option would be to look for formulas within the spreadsheet which if copied and pasted to the cell under consideration would result in the expected value being computed. All such candidate formulas could be ranked from high to low confidence with increasing distance from the target cell.

The number of mutants that have not been corrected by this version of GoalDebug are shown in Table 4. These are the cases in which none of the suggestions generated by GoalDebug correct the mutation.

The evaluation setup is shown in Figure 2. We ran the mutation operators on the sheets and then compared the output from the generated mutants with the output from the original spreadsheet. For the cells in which the outputs from the original spreadsheet and the mutants were different, we specified the outputs from the original spreadsheet as the expected values and ran GoalDebug to generate change suggestions. We then applied the generated change suggestions to the mutated spreadsheets to determine the number of cases in which the mutation is reversed by applying the change suggestion generated by GoalDebug. In

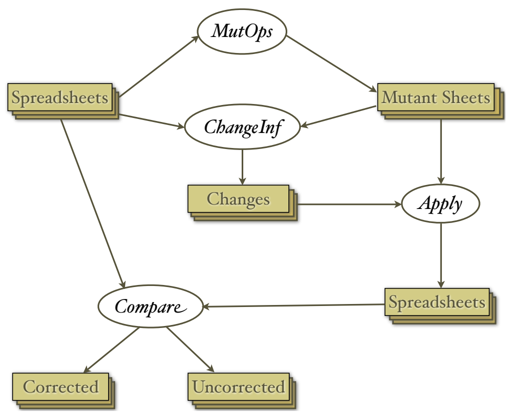

Figure 2. Evaluation setup

cases in which GoalDebug was effective at correcting the mutants, we determined the rank of the change suggestions that would reverse the effect of the mutation operators.

We carried out the evaluation using the suite of mutation operators to get an idea of what kind of extensions are required for GoalDebug to be able to suggest changes for a wide range of faults. It should be clear from a comparison of the different mutation operators in Table 2 and the change suggestion inference mechanism shown in Figure 1 that the original version of GoalDebug would not be able to reverse all the possible mutations. The numbers shown in Table 4 reflect this fact since a majority of the mutations are not reversed by GoalDebug. All of the uncorrected mutants are created by the nine operators shown in Table 4. Moreover, GoalDebug does not have any change inference mechanism to reverse the errors seeded by the operators AOR, CRR, LCR, NRE, NRS, and ROR. To extend GoalDebug's scope, it is important to include coverage for these classes of errors. Moreover, we also would like to improve the system's

$$\begin{aligned}
\delta(a, f(e_1,\ldots,e_k),\gamma) &= \{a : f \rightsquigarrow f' \mid f'(e_1,\ldots,e_k) \xrightarrow{s}\!\!\!\!\rightarrow v \wedge \gamma(v)\} && (5)\\
\delta(a, f(e_1,...,c,...,e_k),\gamma) &= \{a : c \rightsquigarrow \upharpoonleft a' \mid f(e_1,...,\upharpoonleft a',...e_k) \xrightarrow{s}\!\!\!\!\rightarrow v \wedge \gamma(v)\} && (6)\\
\delta(a, \wedge(e_1,\ldots,e_k),\gamma) &= \{a : \wedge \rightsquigarrow \vee \mid \vee(e_1,\ldots,e_k) \xrightarrow{s}\!\!\!\!\rightarrow v \wedge \gamma(v)\} && (7a)\\
\delta(a, \vee(e_1,\ldots,e_k),\gamma) &= \{a : \vee \rightsquigarrow \wedge \mid \wedge(e_1,\ldots,e_k) \xrightarrow{s}\!\!\!\!\rightarrow v \wedge \gamma(v)\} && (7b)\\
\delta(a, e_1\ r\ e_2,\gamma) &= \{a : r \rightsquigarrow r' \mid e_1\ r'\ e_2 \xrightarrow{s}\!\!\!\!\rightarrow v \wedge \gamma(v) \wedge r' \in (\{<,\leq,>,\geq,=,\neq\} - \{r\})\} && (8)\\
\delta(a, f(\upharpoonleft a_1,...,\upharpoonleft a_n),\gamma) &= \{a : f(\upharpoonleft a_1,...,\upharpoonleft a_n) \rightsquigarrow f(\upharpoonleft a_1,...,\upharpoonleft a_n,\upharpoonleft a') \mid \rho(s,a,\upharpoonleft a') \wedge f(\upharpoonleft a_1,...,\upharpoonleft a_n,\upharpoonleft a') \xrightarrow{s}\!\!\!\!\rightarrow v \wedge \gamma(v)\} && (9a)\\
\delta(a, f(\upharpoonleft a_1,...,\upharpoonleft a_n),\gamma) &= \{a : f(\upharpoonleft a_1,...,\upharpoonleft a',...,\upharpoonleft a_n) \rightsquigarrow f(\upharpoonleft a_1,...,\upharpoonleft a_n) \mid \upharpoonleft a' \in \{\upharpoonleft a_1,...,\upharpoonleft a_n\} \wedge f(\upharpoonleft a_1,...,\upharpoonleft a_n) \xrightarrow{s}\!\!\!\!\rightarrow v \wedge \gamma(v)\} && (9b)
\end{aligned}$$

Figure 3. Extensions to Change Inference

coverage on errors seeded by the CRP, RFR, and RRR operators.

From the preliminary evaluation, we identified two areas in which GoalDebug could be improved: (1) The change inference mechanism needs to be expanded to included a wider range of error situations (see Section 4). The modification of the system could potentially result in a much higher number of change suggestions being generated under any given condition. This problem requires us to (2) carry out enhancements to the ranking heuristics so that the system performs better even with the higher number of generated suggestions (see Section 5). The goal of refining the ranking heuristics is to ensure that the *correct* suggestions are assigned high ranks to minimize the effort invested by the users while debugging faults in spreadsheets.

4 Improving Change Inference

Since the original version of GoalDebug could handle only the few formula-level mutations shown in Figure 1, a substantial extension of the change-inference mechanism of GoalDebug was required.

4.1 Extension of Change Inference

The function δ to generate change suggestions has been extended as shown in the definitions given in Figure 3.

If a cell a contains a formula $f(e_1,\ldots,e_k)$, and the constraint on the cell is γ, GoalDebug would use definition (2) to "back propagate" the constraint to the arguments of f, or recommend replacing the entire formula with a value v that satisfies the constraint. The extension, shown in definition (5), also generates recommendations that replace the operator/function f with others that would result in the output value v that satisfies the constraints on the cell. Note that this definition is applicable to binary operators as well. This extension is aimed at increasing GoalDebug's effectiveness against the mutations introduced by the AOR operator.

A constant in a formula that does not match the constraints can be replaced with another constant (as is done to reverse the effect of the CRP operator), or with a reference to a cell whose formula evaluates to a value that satisfies the constraints. (The reference can also be to an input cell whose value satisfies the constraints.) This effect is achieved by definition (6) and is aimed at reversing the errors seeded by the CRR operator.

It has been observed in studies [21] that end users often confuse logical connectors, that is, they use OR when they mean AND and vice versa. The LCR operator has been included in the mutation suite to model this class of errors. The effect of the LCR operator can be reversed by the extension shown in definitions (7a) and (7b).

The ROR operator models the cases in which an end user might chose the wrong relational operator, for example, in an IF statement. Definition (8) reverses the effect of the ROR operator by replacing the relational operator with any of the others that would result in the conditional expression satisfying the constraints.

While specifying the range for an aggregation formula, users might accidentally include extra cells or omit cells that they should have included. These errors are modeled using the NRE and NRS operators, respectively. The current version of the operators carry out the inclusion/exclusion of only one cell for each error seeded. Definition (9a) reverses the effect of a single error seeded by NRS. In this case we need the additional check $\rho(s, a, \upharpoonleft a')$ to ensure that the inclusion of reference $\upharpoonleft a'$ in the formula in a does not introduce a cyclic reference in s. The effect of the NRE operator is reversed by definition (9b).

The RFR operator models errors that occur when users pick incorrect references during the editing of formulas. The original version of GoalDebug's change inference for references limited the suggestions to the immediate neighborhood of the incorrect reference. This approach was based on the assumption that such faults primarily arise from mechanical errors (cases in which users accidentally click the cells in the neighborhood of the one they intended to click). The restriction to the cells in the neighborhood has been removed in the new version of the system.

The RRR operator mutates references within contiguous ranges in aggregation formulas. Like for the RFR operator, the restriction to the cells within the neighborhood of the references in the original formula has been removed in the new version.

4.2 Evaluation of Change Inference

After incorporating the modifications described in Section 4.1, we ran the new version of GoalDebug through the evaluation steps described in Section 3. Once again, the number of mutants that were not reversed by GoalDebug

Sheet	Operators: Uncorrected [Total]								
	AOR	CRP	CRR	LCR	NRE	NRS	RFR	ROR	RRR
Microgen	0 [6]	0 [3]	0 [4]	0 [1]	0 [0]	0 [0]	0 [16]	0 [3]	0 [0]
GradesNew	0 [18]	0 [4]	0 [25]	0 [1]	0 [0]	0 [0]	0 [123]	0 [10]	0 [0]
FitMachine	0 [9]	2 [6]	0 [11]	0 [1]	0 [0]	0 [0]	0 [41]	0 [6]	0 [0]
Digits	0 [62]	0 [17]	0 [43]	0 [0]	0 [0]	0 [0]	0 [152]	0 [19]	0 [0]
NetPay	0 [3]	0 [6]	0 [12]	0 [0]	0 [0]	0 [0]	0 [22]	0 [4]	0 [0]
Purchase	0 [14]	0 [4]	0 [29]	0 [0]	0 [0]	0 [0]	0 [91]	0 [15]	0 [0]
RandJury	0 [87]	0 [47]	0 [33]	0 [6]	0 [0]	0 [0]	0 [119]	0 [16]	0 [0]
Sales	0 [72]	0 [12]	0 [49]	0 [0]	0 [0]	0 [0]	0 [205]	0 [0]	0 [0]
Solution	0 [21]	0 [2]	0 [12]	0 [0]	0 [0]	0 [0]	0 [77]	0 [4]	0 [0]
Budget	0 [15]	0 [1]	0 [18]	0 [0]	0 [0]	0 [0]	0 [75]	0 [3]	0 [0]
MBTI	0 [43]	0 [24]	0 [35]	0 [16]	0 [0]	0 [0]	0 [120]	0 [5]	0 [0]
NewClock	0 [20]	1 [11]	0 [22]	0 [1]	0 [0]	0 [0]	0 [97]	0 [14]	0 [0]
GradesBig	0 [6]	1 [5]	0 [28]	0 [1]	23 [99]	0 [27]	0 [103]	5 [14]	73 [364]
Harvest	0 [0]	0 [0]	0 [5]	0 [0]	0 [40]	0 [18]	0 [24]	0 [0]	10 [134]
Payroll	0 [170]	0 [42]	0 [166]	0 [0]	0 [0]	0 [0]	0 [641]	0 [38]	0 [0]
Total	0 [546]	4 [184]	0 [492]	0 [27]	23 [139]	0 [45]	0 [1906]	5 [151]	83 [498]

Table 5. GoalDebug's effectiveness at correcting mutations after enhancements

was compared against the total number of mutants generated (which remains the same as in the evaluation described in Section 3). The new effectiveness scores are shown in Table 5. Figure 4 shows comparisons of these scores with those from the old system in Table 4. For each of the mutation operator, the percentage coverage of the old system (in yellow/lighter shade) is shown against and the coverage of the new system (in blue/darker shade). The extensions to the change inference mechanism has increased the ability of GoalDebug to recover from a much wider spectrum of errors as can be seen from the plot.

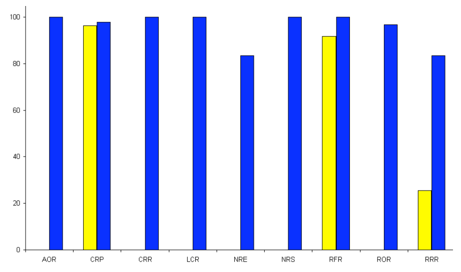

Figure 4. Comparison of coverage%

5 Improving Ranking Heuristics

Since the extension of change inference cause the generation of significantly more change suggestions, the ranking heuristics have to be improved for the system to be effective.

5.1 Extensions to the Ranking Heuristics

Similarities in spreadsheet formulas have been exploited in consistency checking [19] and testing of spreadsheets [10]. The frequency of occurrences of cp-similar regions has been shown by the analysis carried out on the EUSES spreadsheet corpus as reported in [16]. The corpus has 4498 spreadsheets collected from various sources. Out of the 1977 spreadsheets that have formulas in them, 1797 have cp-similar regions. Furthermore, among the spreadsheets that have cp-similar regions, there are on average 5.2 regions per spreadsheet, with an average of 13.1 regions in spreadsheets that have at least 1 region, a maximum of 414 regions in a spreadsheet, and 23845 regions in total in all the spreadsheets taken together.

We define *spatial similarity* as the similarity of formulas in spatially co-located cells. In some cases, cells with similar formulas are not in the immediate spatial neighborhood of each other. Such situations might arise when the cells under consideration are fulfilling similar conceptual roles in different regions of the spreadsheet. To express this idea, we define *conceptual similarity* as the similarity of formulas in cells that are not spatial neighbors. The extensions to the original ranking heuristics are aimed at exploiting the spatial and data flow information in the spreadsheet.

Refinement of the original heuristics. One problem with the original ranking heuristics is that the different kinds of change suggestions are treated on an equal footing. For example, a change suggestion that recommends changing a reference r_1 in a formula f to a reference r_2 (resulting in formula f') is ranked solely on the basis of the Manhattan distance between r_1 and r_2 and the similarity between formulas f and f'. It does not take into account the structure of the formulas in the other cells within the spreadsheet.

In the new heuristics for ranking formula changes, the ranks have three components which are considered for the overall ranking.

1. The similarity (as described in Section 2.1) between the original formula and the suggested new formula.
2. The number of formulas in the spreadsheet that are

similar to the suggested formula.

3. The Manhattan distance of formulas in the spreadsheet that are similar to the suggested formula from the cell the change suggestion is applicable to.

This approach has been adopted to allow the spatial and conceptual neighbors to induce a higher ranking for a formula that is similar to the others within the spreadsheet.

Unit checking. The UCheck system [1, 6] performs automatic consistency checking of spreadsheet formulas based on labels to detect what we call *unit errors*. By integrating UCheck with GoalDebug, we rank unit-correct suggestions higher than other changes that were not unit correct. This integration allows us to add a level of checking to the change suggestion process. Change suggestions are also type checked to ensure that applying any of the change suggestions will not introduce type errors in the spreadsheet.

Impact analysis. We also rank suggestions based on the number of cells that would get affected by applying a particular change. For example, assume that cells c_1, c_2, and c_3 have references to cell c_4, whereas the output of cell c_5 is only used by c_6. Therefore, any change to the formula in c_5 would be preferred over a change to the formula in c_4 since it has lower impact on other cells within the spreadsheet.

User confidence. We assign a level of confidence to the generated change suggestions based on how the user specifies the expected value for a cell. For example, if the expected value in B3 is equal to 80 and the expected value in E7 is less than 90, we assume that the user is more confident about the expected outcome for B3 than for E7. Everything else remaining the same, the change suggestions generated from the constraint on the expected value of B3 would be ranked higher than those generated from the constraint on the expected value of E7. In other words, the confidence level can not only be used for ranking, but also to resolve conflicts while propagated constraints from different sources are encountered.

Exploiting data flow information. When the user indicates that the output of a cell is incorrect, the source of the failure could be a fault in some cell upstream in the dataflow chain from the marked cell, or the fault could be in the marked cell itself. This question can be resolved by seeking more input from the user by shading the cells upstream and asking the user if the values in those cells are correct. Once a cell that has a fault has been isolated, the changes suggested for the formula in that cell should be ranked the highest. Moreover, the change suggestions generated for those cells already marked as correct should be ranked really low—we do not filter these out, just in case the user made a mistake. This approach helps minimize the number of suggestions generated by trying to first locate the fault. This approach is also very effective in cases in which multiple faults lead to the common point of failure that is identified and marked by the user.

5.2 Evaluation of Ranking Heuristics

We have four possible configurations (old and new versions of the change-inference system and ranking heuristics) that can be considered to evaluate the ranking heuristics. However, as can be seen from Table 4, the extended change inference system reverses the effect of all the different classes of mutants, whereas the old version of change inference was only effective for a small subset of possible mutants. Therefore, to evaluate the new ranking heuristics, for each of the mutants reversed by the new change-inference mechanism, we compare the rank assigned by the old version (RankO) of the ranking heuristics against the rank assigned by the new version (RankN).

The Wilcoxon test showed that the new ranking heuristics perform significantly better than the old ones ($p < 0.001$). Ideally, the correct change suggestion should be ranked within the top five ranks, thereby minimizing the effort the user would have to expend to locate it. The difference in ranks assigned by the two techniques is more important at high ranks than at low ones. For example, a difference of 5 between ranks 1 and 6 is more important than between ranks 100 and 105. To account for this aspect, we also ran tests on the reciprocals of the ranks generated by the two techniques. Again, the Wilcoxon test showed that the new ranking techniques perform significantly better than the old ones ($p < 0.001$).

Operator	p
AOR	< 0.001
CRP	< 0.001
CRR	< 0.001
LCR	0.008
NRE	0.036
NRS	0.005
RFR	< 0.001
ROR	< 0.001
RRR	< 0.001

Since the mutation operators reflect different kinds of errors that can occur in formulas, we also compared the performance of the ranking heuristics for each operator. The new heuristics are significantly better than the old ones for all operators as illustrated by the p-values shown on the left. Due to space constraints, we unfortunately cannot show the boxplots.

The cumulative coverage percentages across ranks for the new heuristics (in dark red) are compared against those for the old (in light blue) in Figure 5.

With the new heuristics in effect, the top ranked suggestion corrects the mutations in 59% of the cases, the top two suggestions correct the mutations in 71% of the cases, and so on. Putting the numbers in perspective, out of the 3988 mutants considered, the suggestion that corrects the mutation is ranked in the top five in 80% of the cases with the new ranking heuristics as opposed to only 67% of the cases with the old version of the system.

6 Related Work

The WHYLINE system, implemented in the Alice environment, allows users to ask "Why...?" and "Why didn't...?" questions about expected program behavior [18]. The system uses static and dynamic analyses of the program

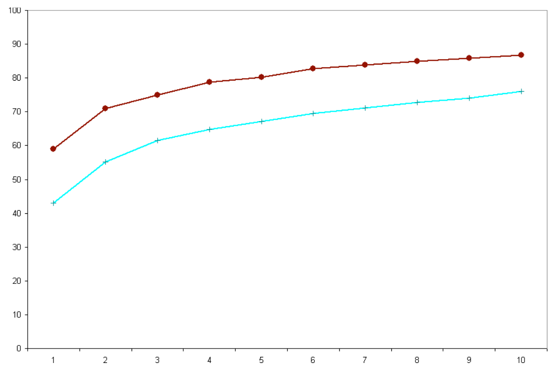

Figure 5. Cumulative coverage of ranking heuristics

to help the user locate the cause of the error, which is similar to the idea of GoalDebug. Empirical evaluations have shown that users debug errors up to 8 times as fast with WHYLINE than without, even though WHYLINE does not produce change suggestions.

Spreadsheet testing is closely related to debugging. In the WYSIWYT system users can indicate incorrect output values by placing a ✗ in the cell. Similarly, they can indicate that the value in a cell is correct by placing a ✓ [26]. When a user indicates one or more program failures during this testing process, fault localization techniques [27] direct the user's attention to cells with possible errors. However, WYSIWYT provides no help with regard to how to change erroneous formulas. It might seem that the GoalDebug analysis subsumes the analysis of WYSIWYT since providing an expected value that is different from the current cell value implies that the current cell value is wrong. However, this is not the case since, in contrast to GoalDebug, WYSIWYT also collects user input about correct cell values and employs this information in the fault localization analysis. Therefore, each system has something to offer to the other.

There are several spreadsheet analysis tools that try to reason about the units of cells to find inconsistencies in formulas [1,6,7,9,13]. The tools differ in the rules they employ and also in the degree to which they require users to provide additional input. The original proposal in [13], which modeled the unit structure essentially with *and* and *or* units arranged into an *is-a* unit hierarchy, was extended by [7] to include an additional *has-a* hierarchy. The approach of [9] is focused on reasoning about dimensions. All these approaches require the user the annotate the spreadsheet cells with additional information. In contrast, the UCheck system [6], by exploiting techniques for automated header inference [1], 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 [22], auditing [19,28], and adoption of better spreadsheet design practices [17,25,31]. Again, none of these approaches offer support for debugging.

7 Conclusions and Future Work

In this paper we have described an evaluation we carried out of our spreadsheet debugger for end users. Guided by the results of our evaluation, we made improvements to the change inference system so that GoalDebug can work with a much wider range of end-user errors. We also made refinements to the ranking heuristics employed by the system so as to minimize the effort expended by the users in locating the correct change suggestion to apply. Further evaluations have shown that the coverage of the system has been increased substantially and that the new ranking heuristics are significantly better than the old ones. The overall effectiveness of the system is quite promising.

In future work, we plan to investigate further possibilities of enhancing the ranking heuristics, in particular, the idea of employing model information about the spreadsheet. The use of unit inference to rank suggestions was one example, but there are other possibilities yet to be explored. For example, the notion of *spreadsheet templates* summarize the very expressive structural aspects of spreadsheets [11,12]. We have already shown that automatic template inference can be performed with high precision and reliability [4]. It would be interesting to see how much template conformance of suggestions can further improve the effectiveness.

Moreover, we would like to carry out user studies to evaluate if end users can use the system effectively. The data from these studies would be invaluable in designing refinements to the system interface. There is the inherent risk that users might not heed their own judgment and simply trust the system to be correct and go with one of the highest ranked suggestion always. At the other extreme, we have a situation in which the user might lose trust in the system and ignore the suggested changes altogether (or not invoke the system at all).

Finally, since testing and debugging are complementary activities, we are interested in merging GoalDebug with the testing environment like WYSIWYT (preferably coupled with an automatic test-case generation system like AutoTest [3]). One particularly interesting aspect, as far as GoalDebug is concerned, is the following idea. Analyses of data collected during empirical studies have shown that some users place ✗ or ✓ based on whether the formula (and not the value as expected by the designers of the system) is right or wrong [24]. Users placing ✓ based on the correctness of the formulas in the spreadsheets would provide important information to the GoalDebug system since we can simply filter out all the change suggestions that recommend changing the formulas marked as correct by the user.

Even in cases in which users only indicate the correctness of a value we can use this information to assign a low rank to change suggestions for the formula in that cell.

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] R. Abraham and M. Erwig. Goal-Directed Debugging of Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 37–44, 2005.

[3] R. Abraham and M. Erwig. AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 43–50, 2006.

[4] R. Abraham and M. Erwig. Inferring Templates from Spreadsheets. In *28th IEEE Int. Conf. on Software Engineering*, pages 182–191, 2006.

[5] R. Abraham and M. Erwig. Mutation Testing of Spreadsheets. 2006. Submitted for publication.

[6] R. Abraham and M. Erwig. UCheck: A Spreadsheet Unit Checker for End Users. *Journal of Visual Languages and Computing*, 2006. To appear.

[7] 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.

[8] C. Allwood. Error Detection Processes in Statistical Problem Solving. *Cognitive Science*, 8(4):413–437, 1984.

[9] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the Unit Correctness of Spreadsheet Programs. In *26th IEEE Int. Conf. on Software Engineering*, pages 439–448, 2004.

[10] M. M. Burnett, A. Sheretov, B. Ren, and G. Rothermel. Testing Homogeneous Spreadsheet Grids with the "What You See Is What You Test" Methodology. *IEEE Transactions on Software Engineering*, 29(6):576–594, 2002.

[11] 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.

[12] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein. Gencel — A Program Generator for Correct Spreadsheets. *Journal of Functional Programming*, 16(3):293–325, 2006.

[13] 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.

[14] EuSpRIG. European Spreadsheet Risks Interest Group. `http://www.eusprig.org/`.

[15] M. Fisher, G. Rothermel, D. Brown, M. Cao, C. Cook, and B. Burnett. Integrating Automated Test Generation into the WYSIWYT Spreadsheet Testing Methodology. *ACM Trans. on Software Engineering and Methodology*, 2006. To appear.

[16] M. Fisher II and G. Rothermel. The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanism. In *1st Workshop on End-User Software Engineering*, pages 47–51, 2005.

[17] T. Isakowitz, S. Schocken, and H. C. Lucas, Jr. Toward a Logical/Physical Theory of Spreadsheet Modelling. *ACM Transactions on Information Systems*, 13(1):1–37, 1995.

[18] A. J. Ko and B. A. Myers. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Int. Conf. on Human Factors in Computing Systems*, pages 151–158, 2004.

[19] R. Mittermeir and M. Clermont. Finding High-Level Structures in Spreadsheet Programs. In *9th Working Conference on Reverse Engineering*, pages 221–232, 2002.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination Of Sufficient Mutant Operators. *ACM Trans. on Software Engineering and Methodology*, 5(2):99–118, 1996.

[21] J. F. Pane, C. A. Ratanamahatana, and B. A. Myers. Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *Int. Journal of Human-Computer Studies*, 54:237–264, 2001.

[22] R. R. Panko. Applying Code Inspection to Spreadsheet Testing. *Journal of Management Information Systems*, 16(2):159–176, 1999.

[23] R. R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. In *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.

[24] 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*, pages 45–52, 2005.

[25] K. Rajalingham, D. Chadwick, B. Knight, and D. Edwards. Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development. In *33rd Hawaii Int. Conf. on System Sciences*, pages 1–9, 2000.

[26] 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.

[27] J. Ruthruff, E. Creswick, M. M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main. End-User Software Visualizations for Fault Localization. In *ACM Symp. on Software Visualization*, pages 123–132, 2003.

[28] J. Sajaniemi. Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *Journal of Visual Languages and Computing*, 11:49–82, 2000.

[29] C. Scaffidi, M. Shaw, and B. Myers. Estimating the Numbers of End Users and End User Programmers. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.

[30] G. Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology*, RTI Project Number 7007.011, 2002.

[31] A. G. Yoder and D. L. Cohn. Real Spreadsheets for Real Programmers. In *Int. Conf. on Computer Languages*, pages 20–30, 1994.