Sharing reasoning about faults in spreadsheets: An empirical study

Joseph Lawrance, Robin Abraham, Margaret Burnett, Martin Erwig
Oregon State University
Corvallis, Oregon 97331
{*lawrance,abraharo,burnett,erwig*}*@eecs.oregonstate.edu*

Abstract

Although researchers have developed several ways to reason about the location of faults in spreadsheets, no single form of reasoning is without limitations. Multiple types of errors can appear in spreadsheets, and various fault localization techniques differ in the kinds of errors that they are effective in locating. In this paper, we report empirical results from an emerging system that attempts to improve fault localization for end-user programmers by sharing the results of the reasoning systems found in WYSIWYT and UCheck. By evaluating the visual feedback from each fault localization system, we shed light on where these different forms of reasoning and combinations of them complement — and contradict — one another, and which heuristics can be used to generate the best advice from a combination of these systems.

1. Introduction

Spreadsheet systems like Excel are among the most widely used programming systems, yet up to 90% or more of spreadsheets contain faults [14, 16]. Because spreadsheets are often used for important tasks and decisions, faults in them have been tied to costly errors.[1]

Researchers have been working to address this problem by developing systems to locate faults within spreadsheets and by evaluating the effectiveness of such systems through empirical studies of end users [1, 10, 19, 21].

Several spreadsheet fault localization systems have emerged from the work in this area, with each system specializing in locating particular categories of faults. For example, UCheck finds faults relating to the spatial structure of the spreadsheet, whereas WYSIWYT[2] relies on data-flow relationships within a spreadsheet and on users' debugging decisions to locate faults. Both systems, described later in this paper, shade formula cells considered to have a fault.

Empirical research into spreadsheet fault localization has shown consistently that end users who debug spreadsheets follow the advice of fault localization systems (e.g., [21]). That is, end users consistently debug the darkest-shaded cells in the spreadsheet. Thus, it is important to ensure that the darkest-shaded cells correspond as closely as possible to where the faults *actually* appear.

To evaluate how closely this visual fault localization feedback corresponds to where faults appear in a spreadsheet, researchers have established a quantitative measure of *visual effectiveness* [15, 21]. We score fault localization feedback from zero to five (unshaded to darkest, respectively). To compute visual effectiveness, we subtract the average score of cells with correct formulas from the average score of cells with incorrect formulas. Therefore, fault localization systems that yield higher visual effectiveness scores are more likely to locate faults effectively and lead users toward the actual faults within a spreadsheet.

As Ruthruff et al. [21] showed, any fault localization approach that includes some form of reporting or feedback to a human involves two factors, an *information base* and a *mapping*. An *information base* refers to the type of information used to locate faults. *Mappings* transform information bases into fault localization feedback.

Although spreadsheets are essentially a grid of cells, various information bases can be extracted out of spreadsheets, and each information base can highlight different categories of faults. For example, cells often contain explicit relationships to other cells, in the form of cell references, from which data-flow graphs emerge; these data-flow graphs can be used to identify reference faults[3] [8]. Furthermore, the juxtaposition of row and column headers against cells containing data within spreadsheets typically implies unit information about cells. Unit inference can be used to identify certain types of reference, range, and omission faults [2]. Other information bases supplied by end users can assist fault localization. For example, the values of cells are often expected to fall within certain intervals; by asserting intervals on cells, cells whose values fall outside their intervals can be located [7, 6, 8]. Adding assertions helped significantly with non-reference faults, suggesting that the addition of assertions into the environment fills a need not met effectively by the data-flow

[1] `http://www.eusprig.org/stories.htm`

[2] What You See Is What You Test with fault localization

[3] One classification scheme we have found to be useful in our previous research involves two fault types: reference faults, which are faults of incorrect or missing references, and non-reference faults, which are all other faults.

testing methodology alone [8]. Furthermore, in several domains, particularly finance, it is often the case that two cells within a spreadsheet must add up to the same value; asserting relationships such as equality among groups of cells can be used to audit spreadsheets. Our emerging prototype is based on the assumption that reasoning about faults in only one way is insufficient to locate several different categories of faults effectively.

In this paper, we modeled end-user testing and debugging behavior probabilistically based on what end users actually did in previous studies. We evaluated an emerging prototype which combines the fault localization feedback from two reasoning mechanisms: UCheck and WYSIWYT. Our overall research goal was to add insights to the following question: what heuristics are most effective in selecting and combining feedback, and what types of faults do the heuristics find compared to WYSIWYT and UCheck?

2. Related Work

Recent research has focused on assisting end-user debuggers by communicating with the user through visual devices. Woodstein [23] is a software agent that visually assists users in debugging e-commerce errors. Ko and Myers present the Whyline [12], an "interrogative debugging" device for the event-based programming environment Alice. Other research supports program comprehension and debugging by end users in the spreadsheet paradigm. For example, Igarashi et al. [11] present devices to aid spreadsheet users in data-flow visualization and editing tasks. S2 [22] provides a visual auditing feature in Excel: similar groups of cells are recognized and shaded based upon formula similarity, and are then connected with arrows to show data-flow. This technique builds upon the Arrow Tool, a data-flow visualization device proposed by Davis [9]. Ayalew and Mittermeir [5] present a method of fault tracing based on interval testing and slicing, which is similar to our own work on assertions to help users automatically guard against faults [8]. Some recent research automatically detects certain kinds of errors, such as errors in spreadsheet units [1] and types [4]. Although researchers have studied humans debugging empirically [15, 20], to our knowledge, none have studied how well *shared* reasoning systems interacts with *human* choices and mistakes.

3. Background: Debugging with WYSIWYT

The fault localization system found in WYSIWYT relies on users' judgments to locate formulas containing faults. In the course of developing a spreadsheet, users can communicate a judgment that a cell's value is correct with a checkmark (✓), or that a cell's value is incorrect with an X-mark (✗), as shown in Figure 1. Checkmarks contribute to the "testedness" of the cells according to an adequacy criterion detailed in [17], and a cell's testedness is reflected in border colors along a red-to-blue continuum (in print: light gray to black). The system combines the user's ✓-marks and ✗-marks with the dependencies in the cells' formulas to estimate likelihoods of the fault (erroneous formula) being located in various cells. It colors these cells' interiors in light-to-dark amber (gray) to reflect these likelihoods [19]. Thus, the WYSIWYT fault localization and testing methodology maintains the interactive nature of spreadsheet systems by allowing users to incrementally test spreadsheets as they develop them [18, 17].

For example, in the Gradebook spreadsheet shown in Figure 1, the user checked off cells B6:E6 and F4:F5, because the user judged these cells' values as correct, whereas the user placed an X-mark on cells F3, J3, and I4, because the user judged these cells' values as incorrect. In response, WYSIWYT shaded the interior of cell F3 the darkest, and several more cells with lighter shades. These cell shadings indicate to the user where to look for incorrect formulas.

4. Background: Unit Errors in Spreadsheets

Users often enter headers within their spreadsheets to label the data. For example, in the spreadsheet shown in Figure 2, the header Quiz_1 in B2 indicates that the data in column B is somehow related to "Quiz_1", which is in turn a "Score" (as indicated by the header Score in B1). Headers serve as documentation to help the user remember what the data means.

In the context of the spreadsheet shown in Figure 2, the number 67 in cell B3 is not simply an integer. The row (Amanda) and column (Quiz_1) headers tell us that the cell contains the score Amanda got on the first quiz (presumably in some course). We call headers in their function as labels *units*. Amanda, in turn, has Students as its header, and Quiz_1 in B2 has Scores as its header. These header hierarchies give rise to what we call *dependent units*, which in this case are Students[Amanda] and Scores[Quiz_1]. Since both the row and column headers apply at the same time, the inferred dependent units are combined using the *and operator* (&) to give the *and unit* Students[Amanda]&Scores[Quiz_1] for the number 67 in B3. This inferred unit is treated as an implicit type declaration for the cell B3. The units for the other cells that contain data values can be inferred along similar lines.

The units obtained for the data cells are then used to infer the units for formula cells. For example, cell B6 contains the formula AVERAGE(B3,B4,B5). Its unit is inferred as a combination of the units of the cells participating in the operation. All three cells have Scores[Quiz_1] as a common factor from the column-level header. The components from the row-level headers are Students[Amanda] for B3, Students[Andy] for B4, and Students[Christina] for B5. Since the values in the three cells are added together, the units are com-

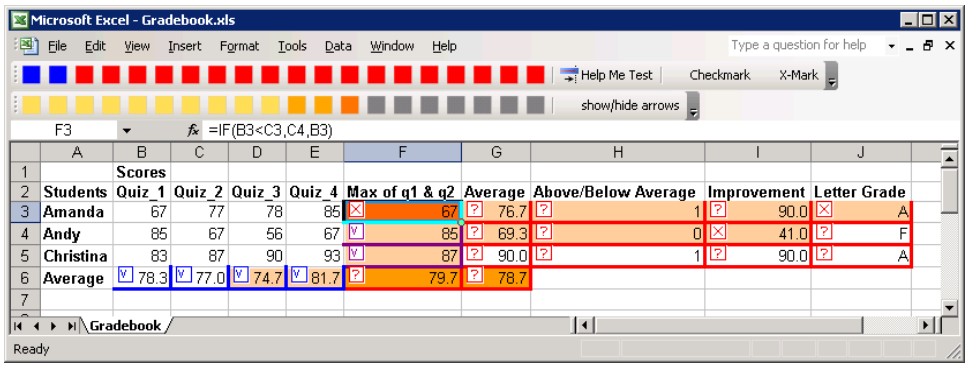

Figure 1. Users' judgments and fault localization feedback using WYSIWYT

bined using the *or operator* (|) to give the *or unit* Students[Amanda]|Students[Andy]|Students[Christina]. The common component can be factored out to yield the unit

Students[Amanda|Andy|Christina].

This unit can be combined with the column-level component using the & operator to give

Scores[Quiz_1]&Students[Amanda|Andy|Christina]

as the unit for B6.

Unit expressions can be combined and transformed according to the formal rule system detailed in [10]. This rule system allows identification of a class of unit expressions that are considered to be well formed. Cell formulas whose derived unit expressions cannot be transformed into well formed units are considered erroneous, and the system reports unit errors for such cells. In the current version of the system, cells that have unit errors are shaded orange. Such errors are called local unit errors. The cells that have formulas that reference cells with unit errors get shaded yellow. Such errors are called propagation unit errors. This fault localization feedback from UCheck is aimed at directing the user's attention to the cells that have primary unit errors since correcting these also removes the propagation unit errors (at least in cases they do not contain their own unit errors).

Consider the following three examples of the errors we seeded in the spreadsheet shown in Figure 2.

The formula in F3 is IF(B3<C3,C4,B3). The inferred unit is Scores[Quiz_2]&Students[Andy] for C4 (if the condition evaluates to True), and the inferred unit is Scores[Quiz_1]&Students[Amanda] for B3 (if the condition evaluates to False). Since the output of the formula could be one or the other, the two units are combined using the or operator (|).[4] UCheck shades the cell F3 orange since the resulting unit is not well formed. Cells G3 and F6 have references to F3, and G6 has a reference to G3. Therefore they are all shaded yellow since the unit error from F3 propagates to cells F6, G3, and G6.

Cells G3 and G5 have the formulas (D3+E3+F3)/3 and (D5+E5+F5)/3, respectively. G4, on the other hand, has the formula (D4+E4)/3. While this formula is in violation of the specifications for the spreadsheet (since the cell is supposed to compute the average of three scores for the student), it is not a unit error. The unit error shows up in G6 whose formula computes the average across the cells G3, G4, and G5 since the inferred unit of G4 (which is missing the component from F4) is incompatible with those of G3 and G5. Cell G6 is shaded yellow in Figure 2 because it also has the propagation unit error from G2. Once the error in F3 is corrected, G6 would only have its local unit error and hence would be shaded orange.

The formula in I3 is
IF(AND(H3<1,B3<C3,C3<D3,D3<E3),G4+10,G5).
The inferred unit for G4 can be reduced to Scores[Quiz_3|Quiz_4]&Students[Andy].
Similarly, the inferred unit for G5 can be reduced to Scores[Quiz_1|Quiz_2|Quiz_3|Quiz_4]&Students[Christina].
These two units are incompatible under the | operation since they are dissimilar on both the Scores and Students components.

Thus, UCheck is a type system that uses the header information entered by the user and assigns units to the values and formulas at a finer level of granularity than the types like Integer and String in traditional programming languages. Instead of detecting and reporting errors the same way as traditional type systems do, UCheck uses the vocabulary of "types" from the user's domain by employing the header information from the spreadsheet the user is working on.

5. A Combined Reasoning System

Our combined reasoning system relies on the results of the independent reasoning systems found in UCheck and in WYSIWYT. As discussed in Sections 3 and 4, the two

[4] The current version of UCheck does not check the units of the operands of the logical comparison for compatibility. That is, in this instance, UCheck does not check the consistency on the units of B3 and C3 for compatibility on < operation.

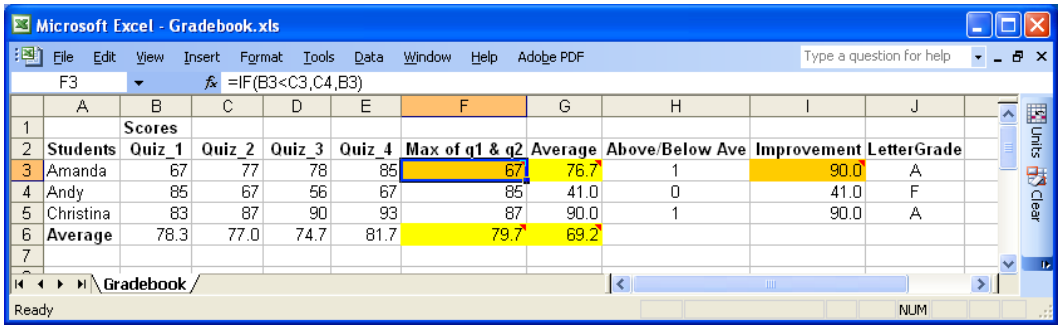

Figure 2. Unit errors in the gradebook spreadsheet as reported by UCheck.

systems base their reasoning on different information bases derived from spreadsheets. While both systems make use of data-flow relationships to locate faults, UCheck analyzes the spatial juxtaposition of row and column headers against data cells, whereas WYSIWYT propagates users' judgments to locate faults.

The architecture of the combined system is shown in Figure 3. In steps 1 and 2, user interactions and Excel spreadsheet information are sent to the reasoning database. In step 3, the spreadsheet cells and users' marks are sent to individual fault localization systems. UCheck carries out unit checking based on the spreadsheet structure and WYSIWYT calculates fault likelihood based on users' judgments. In step 5, the reasoning database collects the results from the two systems. Finally, in steps 6 and 7, the fault localization information for the spreadsheet is computed based on combined results, and the visual feedback in terms of cell shadings is displayed on the spreadsheet. Note that the design depicted in Figure 3 suggests the possibility of including additional reasoning systems in the future; for now, only the reasoning from WYSIWYT and UCheck is used.

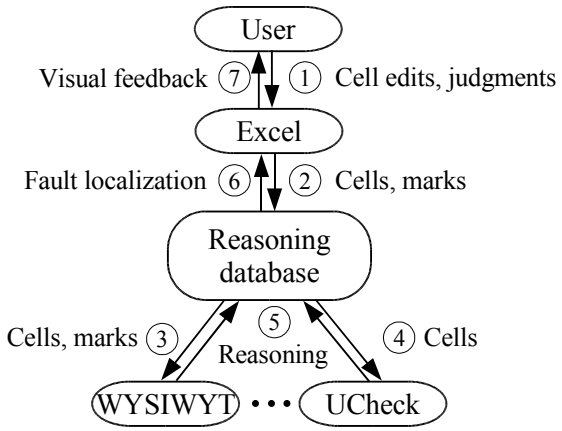

Figure 3. Integrating WYSIWYT and UCheck

In the emerging prototype, we have devised three ways to combine the reasoning from each fault localization system:

- *Combo Max*: return the darkest cell shading received from each fault localization system.
- *Combo Average*: return the "average" of the cell shadings received from each fault localization system.
- *Combo Min*: return the lightest cell shading received from each fault localization system.

6. Experiment

Using our emerging prototype as a testbed, we designed an experiment to evaluate the following research questions:

- RQ1: What heuristics are most effective in combining feedback?
- RQ2: What types of faults do the heuristics find compared to WYSIWYT or UCheck?

6.1. Design

To investigate our research questions, we evaluated our emerging prototype by simulating users who debug spreadsheets. To simulate users, we modeled user behavior based on results from prior empirical work. In these studies, users marked 85% of formula cells on average when testing and debugging spreadsheets, often placing ✓-marks on cells, and rarely placing ✗-marks on cells. Of the cells that users marked, users in our earlier studies made mistakes according to the probabilities given in Table 1, so for our study, we simulated user behavior based on these probabilities. The bold numbers in Table 1 highlight false positive (✓ on incorrect value) and false negative (✗ on correct value) oracle mistakes. Note that even when the value of a cell was incorrect, users were more likely to place a ✓-mark on that cell (false positive) than an ✗-mark. On the other hand, when the value of a cell was correct, users were unlikely to place an ✗-mark on that cell (false negative). Depending on the percentage of cells with incorrect values, users made incorrect testing decisions between 5% to 20% of the time, although the low proability of false negatives means that users' negative

judgments were more accurate, overall [15, 20, 21].

Table 1. Probabilistic User Model

Value	Formula	✓	✗
Incorrect	Incorrect	**74% "dumb"**	26% correct
	Correct	**75% "smart"**	25% correct
Correct	Incorrect	50% correct	**50% "smart"**
	Correct	99% correct	**1% "dumb"**

6.2. Materials

The experiment utilized a spreadsheet similar to those that appear in Figures 1 and 2. The spreadsheet was derived from an Excel spreadsheet of an instructor.

6.3. Dependent Variable and Measures

The *visual effectiveness* of the feedback provided by the shading scheme was computed as it was in [15]. That is, we defined *visual effectiveness* according to the following formula, where *Correct* and *Faulty* are the set of formula cells with correct formulas and incorrect formulas, respectively, and $score(c)$ is the fault localization feedback for cell c:

$$VE = \sum_{c \in Faulty} \frac{score(c)}{|Faulty|} - \sum_{c \in Correct} \frac{score(c)}{|Correct|}$$

6.4. Evaluation Testbed

Figure 4 shows the design of the evaluation testbed and the sequential flow of information among the components. In previous work, we have developed a suite of mutation operators for spreadsheet formulas [3]. The Driver took the error-free grade spreadsheet as input and generated mutant spreadsheets seeded with one faulty formula. For each mutant spreadsheet, the Aggregator sent the spreadsheet information to the reasoning engines, collected the fault localization feedback, and inserted records into the reasoning database. While interacting with WYSIWYT, the Aggregator also modeled user behavior based on the empirical data we have collected from previous studies. We computed the visual effectiveness score over the entire consistency-checking and/or testing cycle based on simulated user behavior for each mutant spreadsheet. For each cell in each mutant spreadsheet, the Aggregator output:

1. The mutation type
2. The oracle judgment
3. Feedback from each reasoning engine
4. Combined fault localization feedback
5. VE score for each type of fault localization feedback

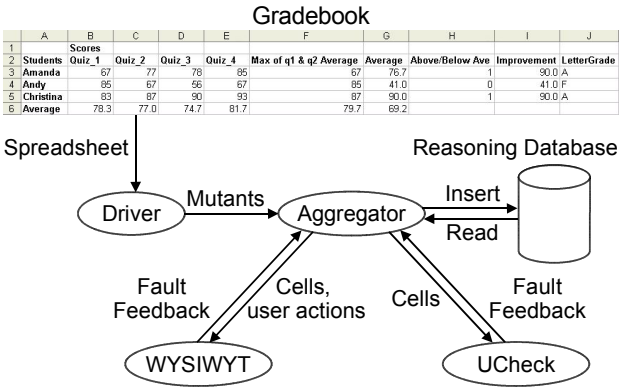

Figure 4. Evaluation testbed architecture

6.5. Procedure

We modeled 932 mutated versions of the `Gradebook` spreadsheet and sent the mutated spreadsheets to the emerging prototype. Each mutated `Gradebook` spreadsheet was seeded with a single mutated formula by applying a mutation operator from the list [3] shown in Table 2. Following established software engineering tradition, we seeded each spreadsheet with a single fault to keep the relationship between the fault being localized and the feedback unambiguous, facilitating our data analysis. The mutations in Table 2 provide coverage of the categories in Panko's classification system [13]. Under Panko's system, mechanical faults include simple typographical errors or wrong cell references. Logical faults are mistakes in reasoning and are more difficult to detect and correct than mechanical faults. An omission fault is information that has never been entered into a cell formula, and is the most difficult to detect [13]. For example, in Table 2, most operations (such as AOR, CRP) provide coverage for mechanical faults, whereas FDL provides some coverage for omission faults, and LCR and ROR provide coverage for logical faults.

For each mutant spreadsheet, the probabilistic model of a user debugged and tested the spreadsheet. We then recorded all feedback from each system and the combined system.

Ruthruff et al. [20] pointed out the impact of mapping on results, so we tried two mappings: one mapping is based on the raw feedback from WYSIWYT and UCheck, another mapping is based on thresholds for each system. In both mappings, we combined the shadings from each system using the three combination strategies given in Section 5. For the *original mapping*, we used the original shadings from each system. For the *threshold mapping*, we ignored propagated unit errors from UCheck and ignored cells with very low bug likelihood from WYSIWYT, then we treated any remaining shaded cells as if they were shaded with the darkest hue.

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

6.6. Threats to validity

Any controlled experiment is subject to threats to validity, and these must be considered in order to assess the meaning and impact of results. Threats to validity are factors other than those accounted for that may be responsible for our results. Wohlin et al. [24] provide a general discussion of validity evaluation and a threats classification.

The specific faults seeded in a spreadsheet can affect fault localization results. To reduce this threat, we used mutation operators providing coverage for categories of faults in Panko's classification scheme [13]. Note, however, that the mutation operators in Table 2 do not all generate the same number of mutants. For example, mutations produced by the CRP, AOR and ROR operators were constrained in their numbers by the number of constants, arithmetic operators, and relational operators (respectively) in the spreadsheet. On the other hand, the RFR operator replaced cell references with references to neighboring cells, thereby generating up to 8 mutants for each reference in the original spreadsheet (we filtered out the mutations that would result in cyclic references or point to empty cells). Since the mutation operators produced different numbers of mutations based on the constraints of the spreadsheet, we tried many different distributions of mutations to get a sense for the robustness of our results, and found that the results were consistent.

Threats to validity also pertain to the extent to which results can be generalized. To increase the representativeness of our spreadsheets, we selected a "real-world" spreadsheet from a real end-user instructor. However, since we did not involve real users in our experiment, it is entirely possible that we overlooked some factor which our probabilistic model did not capture. These validity concerns can be addressed only through repeated studies, using different spreadsheets, faults, and real users.

7. Results

7.1. What heuristics are most effective?

Table 3 shows how the various combination strategies compare to WYSIWYT and UCheck, sorted in descending order of visual effectiveness, for the original mapping (top) and the threshold mapping (bottom), as described in Section 6.5.

Table 3. Shared reasoning vs. UCheck and WYSIWYT

Reasoning	Faulty	-	Correct	=	VE
Combo Max:	1.730	-	0.119	=	1.611
UCheck:	1.526	-	0.042	=	1.484
Combo Average:	1.063	-	0.064	=	0.999
WYSIWYT:	0.600	-	0.085	=	0.515
Combo Min:	0.396	-	0.008	=	0.388
*Combo Max**:	4.043	-	0.166	=	3.877
UCheck*:	3.814	-	0.106	=	3.708
*Combo Average**:	2.339	-	0.093	=	2.246
WYSIWYT*:	0.864	-	0.079	=	0.785
*Combo Min**:	0.636	-	0.019	=	0.617

We performed paired t tests to analyze differences among the reasoning strategies shown in Table 3. To begin, we state the following null hypotheses:

H1: The visual effectiveness of the combined reasoning does not differ from the visual effectiveness of WYSIWYT.

H2: The visual effectiveness of the combined reasoning does not differ from the visual effectiveness of UCheck.

H3: The visual effectiveness of the original mapping does not differ from the visual effectiveness of the threshold mapping.

We found significant differences in the visual effectiveness scores among *Combo Min*, *Combo Max*, *Combo Average*, WYSIWYT, and UCheck, as shown in Table 4; all were significant at the $p < .001$ level. Thus, Table 4 gives evidence to reject both H1 and H2. The t tests in Table 4 for H1 and H2 are symmetrical, but are repeated for clarity. These results give evidence to suggest that the combination of feedback from UCheck and WYSIWYT is better than either system alone.

We found significant differences in the visual effectiveness between the original mapping and the threshold mapping, as shown in Table 4. Thus, Table 4 gives evidence to reject H3. Modifying the visual feedback through a threshold was sufficient to improve the effectiveness of UCheck, WYSIWYT, and the combinations, in turn. This corroborates the importance of the visual mapping factor in the effectiveness of fault localization systems [20]. Additional tests demonstrated that the same statistically significant differences in the visual effectiveness scores among

WYSIWYT, UCheck, and the combinations also held for the threshold mapping. Therefore, the relative visual effectiveness of WYSIWYT, UCheck, and the combinations are robust to changes in mapping.

Table 4. Significance tests (df = 931, p $<$ 0.001)

	System	System	t
H1	**WYSIWYT**	*Combo Min*	**5.36**
	WYSIWYT	***Combo Average***	**−20.23**
	WYSIWYT	***Combo Max***	**−26.95**
H2	**UCheck**	*Combo Min*	**26.95**
	UCheck	*Combo Average*	**20.23**
	UCheck	***Combo Max***	**−5.36**
H3	***Combo Max****	*Combo Max*	**33.07**
	UCheck*	UCheck	**34.56**
	Combo Average*	*Combo Average*	**29.52**
	WYSIWYT*	WYSIWYT	**5.65**
	Combo Min*	*Combo Min*	**5.36**

7.2. What classes of faults are detected by each system?

There has been a little research into classes of faults detected by WYSIWYT and UCheck individually [1, 21]. We want to add to this growing body of knowledge and also to investigate how the two systems' strengths might complement each other. To investigate this question, we tested the following null hypothesis:

H4: The fault type is unrelated to the effectiveness of WYSIWYT only, UCheck only, or both systems.

Statistical analysis causes us to reject H4. The feedback provided by UCheck only, WYSIWYT only, and both systems is dependent on the type of fault ($\chi^2 = 8897.803, df = 27, p < 0.001$). Thus, WYSIWYT and UCheck differ in their ability to uncover various types of faults.

Table 5 summarizes the relationship between the observed faults and the feedback provided by UCheck only, WYSIWYT only, and both systems. Whereas UCheck was highly effective at detecting a narrow set of faults (particularly NRE and RRR mutations), WYSIWYT was rather effective at detecting a broader range of faults, as suggested by the "T" shape depicted in Table 5. In this way, UCheck and WYSIWYT complement each other.

The Venn diagrams in Figure 5 illustrate how often UCheck, WYSIWYT, and their combinations shaded faulty formulas and correct formulas. The intersection of UCheck and WYSIWYT corresponds to the formulas shaded by *Combo Min*, whereas the union of UCheck and WYSIWYT corresponds to the formulas shaded by *Combo Max*. Only 73 faulty formulas were not shaded by either system; in contrast, 12119 correct formulas were not shaded by either system.

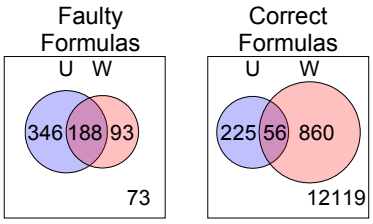

Figure 5. UCheck (U) and WYSIWYT (W)

8. Discussion and Implications

The broader range of faults that WYSIWYT detected might explain how *Combo Max* was able to achieve the highest visual effectiveness score in our analysis — even though the *Combo Max* heuristic was more likely to shade correct formulas than either system alone (as shown in Figure 5).

In contrast to *Combo Max*, *Combo Min* is very conservative. Since *Combo Min* was much less likely to shade correct formulas than either system (shown in Figure 5), *Combo Min* was the most trustworthy form of feedback identified in our analysis.

As the results of the study show, fault detection through the use of units and testing complement each other. UCheck is very effective against a narrow range of faults, whereas WYSIWYT — due to its reliance on users who make mistakes — is less effective for some fault types, but locates a broader range of faults than UCheck. Each excels at particular types of faults, and each overlooks other types of faults. Given effective combination heuristics, each can help the other overcome limitations.

Even more important, the combination supports different human work styles. To use UCheck to its best advantage, the user's time investment is concentrated mostly up front, in the structuring of the spreadsheet under well organized labels. The better job the user does at this, the better UCheck performs on many fault types, leaving fewer that must be found by testing later. On the other hand, if a user is not proficient at labeling, or wants to put off labeling for later, the user can invest less time up front in structuring the spreadsheet in favor of more time later in testing it.

9. Conclusion

In this paper, we showed that reasoning about faults in only one way is not as effective as shared reasoning. In doing so, we also corroborated previous findings that demonstrated the importance of mapping in fault localization feedback.

More importantly, we demonstrated that the combination of fault localization feedback is beneficial and flexible enough to support different work styles and different design objectives for fault localization systems. Users who invest time into structuring their spreadsheets up front reap the most benefit most UCheck, while users may favor spend-

Table 5. Observed Faults vs. Reasoning

Contributor	CRP	AOR	CRR	RFR	NRE	RRR	NRS	ROR	LCR
Neither	0	2	11	35	0	7	14	1	3
WYSIWYT only (WYSIWYT - UCheck)	3	7	13	50	0	2	13	5	0
Both (WYSIWYT ∩ UCheck)				25	38	125			
UCheck only (UCheck - WYSIWYT)				26	85	235			

ing more time testing by making use of WYSIWYT. Additionally, designers of fault localization systems who want to provide conservative feedback may favor *Combo Min*, while desginers who want to provide highly effective feedback would favor *Combo Max*. Thus, for users and designers of fault localization systems, combined reasoning provides improvements in both accuracy of reasoning and flexibility of use.

Acknowledgments

This work was supported in part by the EUSES Consortium via NSF grant ITR-0325273.

References

[1] Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.

[2] Robin Abraham and Martin Erwig. How to communicate unit error messages in spreadsheets. In *WEUSE I: Proceedings of the first workshop on End-user software engineering*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[3] Robin Abraham and Martin Erwig. Mutation testing of spreadsheets. 2006. Submitted.

[4] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A type system for statically detecting spreadsheet errors. In *Proc. IEEE Conf. Auto. Soft. Eng.*, 2003.

[5] Y. Ayalew and R. Mittermeir. Spreadsheet debugging. In *Proc. European Spreadsheet Risks Interest Group*, 2003.

[6] Yirsaw Ayalew. *Spreadsheet Testing Using Interval Analysis*. PhD thesis, Universität Klagenfurt, 2001.

[7] Yirsaw Ayalew, Markus Clermont, and Roland Mittermeir. Detecting errors in spreadsheets. In *Proceedings of EuSpRIG 2000 Symposium: Spreadsheet Risks, Audit and Development Methods*, 2000.

[8] Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *International Conference on Software Engineering*, pages 93–103, 2003.

[9] J. S. Davis. Tools for spreadsheet auditing. *Int. J. Human-Computer Studies*, 45:429–442, 1996.

[10] M. Erwig and M. Burnett. Adding apples and oranges. In *4th Int. Symp. on Practical Aspects of Declarative Languages*, pages 173–191, 2002.

[11] T. Igarashi, J. D. Mackinlay, B. W. Chang, and P. T. Zellweger. Fluid visualization of spreadsheet structures. In *Proc. IEEE Symp. Visual Langs.*, pages 118–125, 1998.

[12] A. J. Ko and B. A. Myers. Designing the whyline: A debugging interface for asking questions about program failures. In *Proc. ACM Conf. Human Factors Computing Systems*, pages 151–158, 2004.

[13] R. Panko. What we know about spreadsheet errors. *Journal of End User Computing*, 10(2):15–21, 1998.

[14] Raymond R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. In *Proceedings of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.

[15] Amit Phalgune, Cory Kissinger, Margaret Burnett, Curtis Cook, Laura Beckwith, and Joseph R. Ruthruff. Garbage in, garbage out? An empirical look at oracle mistakes by end-user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005.

[16] K. Rajalingham, D. R. Chadwick, and B. Knight. Classification of spreadsheet errors. In *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2001.

[17] Gregg Rothermel, Margaret Burnett, Lixin Li, Christopher Dupuis, and Andrei Sheretov. A Methodology for Testing Spreadsheets. *ACM Trans. Software Engineering and Methodology*, 10(1):110–147, 2001.

[18] Karen J. Rothermel, Curtis R. Cook, Margaret M. Burnett, Justin Schonfeld, T. R. G. Green, and Gregg Rothermel. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In *ICSE '00: 22nd International Conf. Software Engineering*, pages 230–239, 2000.

[19] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main. End-user software visualizations for fault localization. In *Proceedings of ACM Symposium on Software Visualization*, pages 123–132, 2003.

[20] Joseph R. Ruthruff, Margaret Burnett, and Gregg Rothermel. An empirical study of fault localization for end-user programmers. In *International Conference on Software Engineering*, 2005.

[21] Joseph R. Ruthruff, Shrinu Prabhakararao, James Reichwein, Curtis Cook, Eugene Creswick, and Margaret Burnett. Interactive, visual fault localization support for end-user programmers. *Journal of Visual Languages and Computing*, 16(1-2):3–40, 2005.

[22] J. Sajaniemi. Modeling spreadsheet audit: A rigorous approach to automatic visualization. *J. Visual Langs. Computing*, 11(1):49–82, 2000.

[23] E. J. Wagner and H. Lieberman. Supporting user hypotheses in problem diagnosis on the web and elsewhere. In *Proc. Int. Conf. Intelligent User Interfaces*, pages 30–37, 2004.

[24] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.