*Technical Report 04-60-05 — School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA, 29 March 2004*

# Interactive, Visual Fault Localization Support for End-User Programmers*

Joseph R. Ruthruff[†], Shrinu Prabhakararao[†], James Reichwein[‡], Curtis Cook[†], Eugene Creswick[†], and Margaret Burnett[†]
*School of Electrical Engineering and Computer Science*
*Oregon State University, Corvallis, Oregon, 97331, USA*
[†]{*ruthruff, prabhash, cook, creswick, burnett*}*@cs.orst.edu*, [‡]*jreichwe@san.rr.com*

**Abstract**

*End-user programmers are writing an unprecedented number of programs, primarily using languages and environments that incorporate a number of interactive and visual programming techniques. To help these users debug these programs, we have developed an entirely visual, interactive approach to fault localization. This paper presents the approach. We also present the results of a think-aloud study that examined the interactive, human-centric issues that arise in end-user debugging using a fault localization strategy. Our results provide insights into the contributions such strategies can make to the end-user debugging process.*

**Keywords:** end-user programming, visual fault localization, debugging, end-user software engineering, testing, slicing, form-based visual programs

## 1 Introduction

Recent years have seen the explosive growth of end-user programming. In fact, by the year 2005, it is estimated that there will be approximately 55 million end-user programmers in the U.S. alone, as compared to an estimated 2.75 million professional programmers [9]. Real-world examples of end-user programming environments include educational simulation builders, web authoring systems, multimedia authoring systems, e-mail filtering rule systems, CAD systems, scientific visualization systems, and spreadsheets. These systems all make use of at least some visual programming techniques in order to support their users, such as graphical syntaxes, drag-and-drop to specify desired outcomes or appearances, programming by demonstration, and/or immediate visual feedback about a program's semantics.

But, how reliable are the programs end users write using such systems? One of the most widely used real-world end-user programming paradigms is the spreadsheet. Despite its perceived simplicity, evidence from this paradigm reveals that end-user programs often contain faults [23, 43, 44]. (Following standard terminology [4], in this paper, a *failure* is an incorrect computational result, and a *fault* is the incorrect part

*This paper updates and extends earlier work that appeared in [47] and [49].

of the program that caused the failure.) Perhaps even more disturbing, users regularly express unwarranted confidence in the quality of these programs [20, 44].

To help solve the reliability problem, we have been working on a vision we call "end-user software engineering" [16]. The concept of end-user software engineering is a holistic approach to the facets of software development in which end users engage. Its goal is to bring some of the gains from the software engineering community to end-user programming environments, *without* requiring end users to have training, or even interest, in traditional software engineering concepts or techniques. Our end-user software engineering devices communicate with their users entirely through interactive, visual mechanisms.

We are prototyping our end-user software engineering methodologies in the *form-based paradigm* because it is so widespread in practice. Form-based languages provide a declarative approach to programming, characterized by a dependence-driven, direct-manipulation working model [3]. The form-based paradigm is inherently visual because of its reliance on multiple dimensions. In form-based languages, the programmer designs a form in two or more dimensions, including formulas which will ultimately compute values. Each formula corresponds to the right-hand side of an equation, i.e., $f(Y_1, Y_2, \ldots, Y_n)$, and is also associated with a mechanism to visually display the formula's value. For example, in spreadsheet languages, a cell is the mechanism that associates a formula with a display mechanism. When a cell's formula is defined, the underlying evaluation engine calculates the cell's value and those of other affected cells (at least those that are visible to the user), and displays new results. Examples of this paradigm include not only commercial spreadsheet systems, but also research languages used for purposes such as producing high-quality visualizations of complex data [19], for specifying full-featured GUIs [40, 57], for matrix manipulation [2, 45, 61], for providing steerable simulation environments for scientists [15], for web programming [12], and for working visually with user-defined objects [11, 14].

In this paper, we introduce a new end-user software engineering device—interactive fault localization—an entirely interactive, visual approach to fault localization that we have integrated into our end-user software engineering concept. Our approach has been carefully designed around five properties, chosen after consideration of both software engineering and human-computer interaction principles. Although we have prototyped our ideas in the form-based paradigm, our design allows our approach to be incorporated into other types of visual programming environments.

We also describe a think-aloud study that we conducted to evaluate how our approach impacts the debugging efforts of end users, and to examine the interactive, human-centric issues that arise in end-user debugging using visual fault localization. The results of this study indicate that our visual fault localization approach can favorably affect the debugging efforts of end users working in form-based visual programming environments. In addition, our study yields several insights into the interactive, end-user debugging process that future work on end-user debugging may need to consider.

The remainder of this paper is organized as follows: Section 2 discusses previous research that is related to fault localization; Section 3 describes the end-user software engineering devices with which our approach to fault localization is integrated; Section 4 introduces our interactive, visual approach to fault localization; Section 5 describes the procedures of the aforementioned think-aloud study; Section 6 outlines the results of the study; Section 7 discusses the implications of our study's results in further detail; Section 8 discusses threats to the validity of our study's results; Section 9 discusses integrating our approach into other programming paradigms; and Section 10 presents our conclusions.

# 2 Related Work

Our approach to interactive fault localization is a visual debugging device intertwined with an environment that emphasizes software engineering practices. Because of this, we look first at the area of software engineering devices that make use of visualizations for debugging. The FIELD environment was aimed primarily at program comprehension as a vehicle for both debugging and instruction [51]. This work draws from and builds upon earlier work featuring visualizations of code, data structures, and execution sequences, such as PECAN [50], the Cornell Program Synthesizer [59], and Gandalf [41]. ZStep [37], on the other hand, aims squarely at debugging and providing visualizations of the correspondences between static program code and dynamic program execution. Its navigable visualizations of execution history are representative of similar features found in some visual programming languages such as KidSim/Cocoa/Stagecast [29] and Forms/3 [5, 11]. An example of visualization work that is especially strong in low-level debugging such as memory leaks and performance tuning is PV [33]. Low-level visualization work specifically aimed at performance debugging in parallel programming is surveyed by Heath [28]. Finally, Eick's work focuses on high-level views of software, mostly with an eye to keeping the bugs under control during the maintenance phase [25].

Work aimed particularly at aiding end-user programmers with debugging and other software engineering tasks is beginning to emerge. Myers and Ko present the Whyline [34], an "Interrogative Debugging" device for the 3D programming environment Alice. Users pose questions in the form of "Why did..." or "Why didn't..." that the Whyline answers by displaying a visualization of a partial program slice that the user can navigate, expand, and compare with previous questions. Igarashi et al. present devices to aid spreadsheet users in dataflow visualization and editing tasks [30]. S2 [56] provides a visual auditing feature in Excel 7.0: similar groups of cells are recognized and shaded based upon formula similarity, and are then connected with arrows to show dataflow. This technique builds upon the Arrow Tool, a dataflow visualization device proposed by Davis [24]. Carr proposes reMIND+ [18], a visual end-user programming language with support for reusable code and type checking. reMIND+ also provides a hierarchical flow diagram for increased program understanding. Outlier finding [39] is a method of using statistical analysis and interactive techniques to

direct end-user programmers' attention to potentially problematic areas during automation tasks. Miller and Myers use visual cues to indicate abnormal situations while performing search and replace or simultaneous editing tasks. Because not all outliers are incorrect, the approach uses a heuristic to determine the outliers to highlight for the user. Finally, the assertions approach in Forms/3 has been shown empirically to help end-user programmers correct errors [13, 63].

Ayalew and Mittermeir present a method of fault tracing for spreadsheets based on "interval testing" and slicing [6]. In their approach, which is similar to assertions in Forms/3 [13], user-specified intervals are compared with cell values and system-generated intervals for each cell. When the user-specified and system-generated intervals for a cell do not agree with the actual spreadsheet computation, the cell is flagged as displaying a "symptom of a fault". Furthermore, upon request from the user, a "fault tracing" strategy can be used to identify the "most influential faulty cell" from the cells perceived by the system to contain symptoms of faults. (In the case of a tie, one cell is arbitrarily chosen.) There are many differences between their approach and our fault localization strategy; we describe two such differences. First, unlike the approach of Ayalew and Mittermeir, our fault localization strategy includes robustness features against user mistakes (described in Section 4.3.1). Second, our approach explicitly supports an incremental end-user debugging process by allowing users to make decisions that can have the effect of improving our continuously-updated feedback.

Fault localization is a very specific type of debugging support. Behind the scenes, most fault localization research has been based on slicing and dicing techniques. (We discuss the details of slicing and dicing in Section 4.2.) A survey of these techniques was made by Tip [60]. Agrawal et al. have built upon these techniques for traditional languages in xSlice [58]. xSlice is based upon displaying dices of the program relative to one failing test and a set of passing tests. Tarantula utilizes testing information from all passing and failing tests to visually highlight possible locations of faults [31]. Both these techniques are targeted at the professional programmer, and return results after running a batch of tests. Besides being explicitly targeted at end users, our methods differ from their approaches in that our methods are interactive and incremental.

Pan and Spafford developed a family of heuristics appropriate for automated fault localization [42]. They present a total of twenty different heuristics that they felt would be useful. These heuristics are based on the program statements exercised by passing and failing tests. Our strategy directly relates to three of these heuristics: the set of all cells exercised by failed tests, cells that are exercised by a large number of failed tests, and cells that are exercised by failing tests and that are not exercised by passing tests.

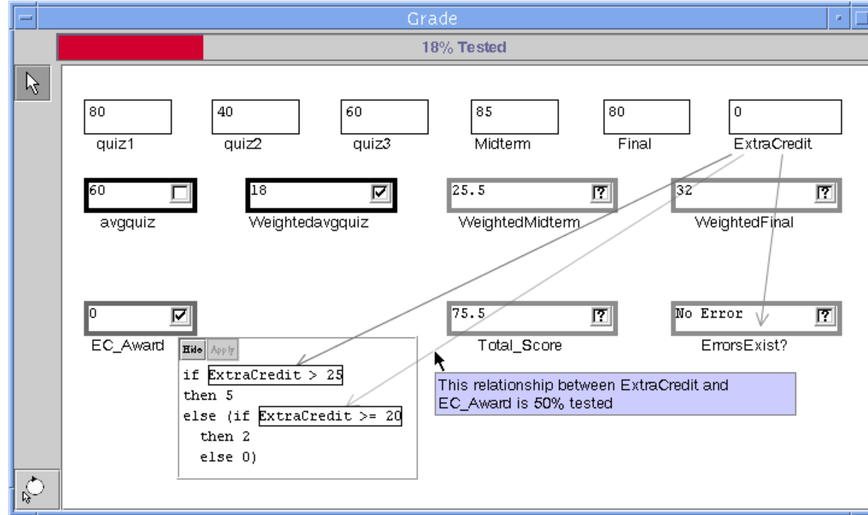


Figure 1: An example of WYSIWYT in the Forms/3 language.

# 3 Background: End-User Software Engineering

We believe that interactive and visual programming techniques, whose support has enabled end-user programming to evolve into a widespread phenomenon, can help reduce the growing reliability problem in programs created by end users. Towards this end, our "end-user software engineering" vision consists of a blend of components that come together seamlessly through interactive visual devices. Components that have been implemented in Forms/3 [11, 14]—a form-based visual programming language—include: "What You See Is What You Test" (WYSIWYT) [17, 52, 53], a visual testing methodology for form-based visual programs; an automated test case generation device [26]; a test re-use strategy [27]; and an approach for supporting assertions by end users [13].

Visual fault localization has been blended into this combination by marrying it to the WYSIWYT visual testing methodology. Therefore, we briefly describe WYSIWYT here.

The underlying assumption behind the WYSIWYT testing methodology is that, as a user incrementally develops a form-based visual program, he or she can also be testing incrementally. The system communicates with the user about testing through visual devices. Figure 1 presents an example of WYSIWYT in Forms/3. In WYSIWYT, untested cells that have non-constant formulas are given a red border (light gray in this paper), indicating that the cell is untested. (Cells whose formulas are simply constants do not participate in WYSIWYT devices, since the assumption is that they do not need to be tested.) For example, the `Total_Score` cell has never been tested; hence, its border is red (light gray). The borders of such cells remain red until they become more "tested".

In order for cells to become more tested, tests must occur. But tests can occur at any time—intermingled with editing formulas, adding new formulas, and so on. The process is as follows. Whenever a user notices a correct value, she can place a checkmark (√) in the decision box at the corner of the cell she observes to be correct: this *testing decision* constitutes a successful "test". Such checkmarks increase the "testedness" of a cell, which is reflected by adding more blue to the cell's border (more black in this paper). For example, in Figure 1, the `Weightedavgquiz` cell has been given a checkmark, which is enough to fully test this cell, thereby changing its border from red to blue (light gray to black). Further, because a correct value in a cell $C$ depends on the correctness of the cells contributing to $C$, these contributing cells participate in $C$'s test. Consequently, in this example the border of cell `avgquiz` also turns blue (black).

Although users may not realize it, the "testedness" colors that result from placing checkmarks reflect the use of a dataflow test adequacy criterion that measures the interrelationships in the source code that have been covered by the users' tests. A cell is fully tested if all its interrelationships have been covered; if only some have been covered then the cell is partially tested. These partially tested cells would have borders in varying shades of purple (shades of gray). (Details of the test adequacy criterion are given in [52, 53].) For example, the partially tested `EC_Award` cell has a purple (gray) border.

In addition to providing feedback at the cell level, WYSIWYT gives the user feedback about testedness at two other granularities. A percent testedness indicator provides testedness feedback at the program granularity by showing a progress bar that fills and changes color from red to blue (following the same colorization continuum as cell borders) as the overall testedness of the program increases. The testedness bar can be seen at the top of Figure 1; the tested indicator shows that the program is 10% tested. Testedness feedback is also available at a finer granularity through dataflow arrows. In Figure 1, the user has triggered the dataflow arrows for the `ExtraCredit` cell. These arrows are different from other form-based visual programming systems' dataflow arrows in two ways. First, they can depict dataflow relationships between subexpressions within cell formulas, as displayed with the `EC_Award` formula in Figure 1. Second, when the user activates the dataflow arrows for a cell, the arrows connecting the formulas' subexpressions follow the same red-to-blue color continuum of cell borders at the granularity of subexpressions. This has the effect of showing users the untested cell reference combinations that still need testing. In addition to color-coding the arrows, the system provides testedness feedback through an intelligent explanation system. In Figure 1, the user has chosen to examine one of the purple (gray) subexpression arrows leading into `EC_Award`, which shows that the relationship between `ExtraCredit` and the indicated *subexpression* of `EC_Award` is 50% tested.

WYSIWYT has been empirically shown to be useful to both programmers and end users [36, 54]; however, it does not by itself explicitly support a debugging effort to localize the fault(s) causing an observed failure. Providing this debugging support is the aim of our interactive, visual fault localization strategy, which we describe next.

# 4 Visual Fault Localization

Testing and debugging are closely related: testing is detecting the presence of failures, and debugging is tracking the failure down to its fault (fault localization) and then fixing the fault. Taking advantage of this relationship, in our approach, the WYSIWYT testing methodology is used as a springboard to fault localization.

Here is the way this springboard works. In addition to the possibility of noticing that a cell's value is correct, there is also the possibility of noticing that a cell's value is incorrect. In the first case, the user checks off the cell's value via WYSIWYT's decision box, as we explained in Section 3. The second case is where fault localization comes in. In that case, which happens when the user has detected a failure, the user can "X out" the value by placing an X-mark in the decision box instead. These X-marks trigger a fault likelihood calculation for each cell with a non-constant formula that might have contributed to the failure. This likelihood, updated for each appropriate cell after any testing decision or formula edit, is represented by visually highlighting the interior of suspect cells in different shades of red. As the fault likelihood of a cell grows, the suspect cell is highlighted in increasingly darker shades of red. The darkest cells are thus estimated to be the most likely to contain the fault, and are therefore the best candidates for the user to consider in trying to debug. Of course, given the immediacy of visual feedback that is customary in form-based visual programming environments, the visual highlighting is added or adjusted immediately whenever the user adds another checkmark or X-mark.

For example, suppose that after working awhile, the user has gotten the `Grades` program to the stage shown at the top of Figure 2. At that point, the user notices that the output value of the `Total_Score` cell is incorrect: the total score is obviously too high. Upon spotting this failure, the user places an X-mark in the `Total_Score` cell, triggering fault likelihood calculations for all cells whose values dynamically contributed to the value in `Total_Score`, and causing the interiors of cells suspected of containing faults to be highlighted in red (gray in this paper), as shown at the bottom of Figure 2. The cells deemed (by the system) most likely to contain the fault are highlighted the darkest.

## 4.1 Design constraints

The above example rests upon a number of constraints on the kind of approach that can be viable for end-user programmers working in a form-based language. We now make these constraints explicit:

1. **Form-based visual programs are modeless.** Form-based visual programming does not require the separate code, compile, link, and execute modes typically required by traditional programming languages. Developers simply draw and manipulate live calculation objects on forms and continually observe how the results seem to be working out. Thus, in order for testing and debugging techniques

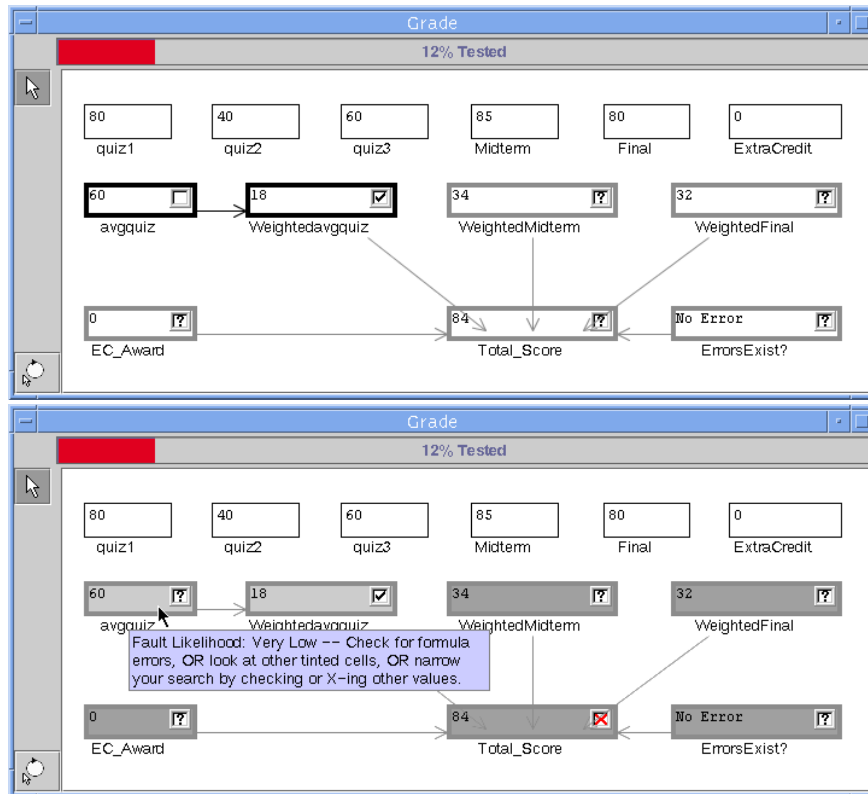


Figure 2: (Top) A `Grades` program at an early stage of testing. (Bottom) The user notices an incorrect output value in `Total_Score` and places an X-mark in the cell. All cells that could have dynamically contributed to this incorrect value have been highlighted in shades of red (gray in this paper), with darker shades corresponding to increased fault likelihood. In this bottom example, `avgquiz` and `Weightedavgquiz` have an estimated fault likelihood of "Very Low" (the fault likelihood of `avgquiz` is indicated by the explanation tooltip) while `EC_Award` and the four rightmost cells have a fault likelihood of "Low" (not indicated by tooltips). Finally, the testedness borders of cells with non-zero fault likelihood have faded, leaving visual focus on the fault likelihood component of the cell.

to be consistent with this paradigm, the users must be allowed to debug and test incrementally and modelessly, in parallel with program development.

2. **Form-based visual program developers are not likely to understand testing and debugging theory.** For an end-user audience, testing and debugging techniques cannot depend on the user understanding testing or debugging theory, nor should such techniques rely on specialized vocabularies based on such theory.

3. **Form-based visual programming environments must maintain trust with their end-user developers.** End-user programmers may not understand the reasons if debugging feedback leads them astray. Instead, they are likely to lose trust in our methodology and ignore the feedback. (As [21] explains, trust is critical to users actually believing a system's output.) Therefore, our methodology must maintain some consistent level of guarantees about its correctness, so that users can understand to what extent they can believe its feedback.

4. **Form-based visual programs offer immediate feedback.** When an end-user programmer changes a formula, the environment displays the results quickly. Users have come to expect this responsiveness from form-based visual programs and may not accept functionality that significantly inhibits responsiveness. Therefore, the integration of testing and debugging into these programming environments must minimize the overhead it imposes, and must be interactive and incremental.

One direct result of these constraints is that many preexisting fault localization techniques that are often designed for imperative programs, in addition to techniques performing a batch processing of information, are not suited towards the audience we wish to target. For example, batch-oriented algorithms for fault localization that expect all testing to be completed before any fault localization processing can be performed inherently cannot provide the incremental, immediate feedback expected in the form-based visual programming paradigm. An entirely new approach to fault localization is therefore necessary.

## 4.2 Slicing and dicing

Our approach performs its calculations leading to the visual representation of fault likelihood by drawing from information gleaned via slicing, and by making use of that information in a manner inspired by dicing. This information is used to incrementally calculate the colorizations of our fault localization approach so that they can be interactively updated after each user action.

Weiser introduced *program slicing* [62] as a technique for analyzing program dependencies. A slicing criterion $<p, V>$, where $p$ is a program point and $V$ is a subset of program variables, is used to define a program slice. Program slices can then be classified into two categories: backward slicing finds all the

program points $P$ that affect a given variable $v$ at a given program point $p$, whereas forward slicing finds all the program points $P$ that are affected by a given variable $v$ at a given program point $p$. (In Forms/3, cells are considered variables. Consequently, a cell $A$ is in the backward slice of a cell $B$ if $B$ is dependent on $A$—meaning the value of $A$ affects the output of $B$. Conversely, in the preceding example, $B$ is said to be in the forward slice of $A$.) Weiser's slicing algorithm calculates *static* slices, based solely on information contained in source code, by iteratively solving dataflow equations.

*Dynamic slicing* [35] uses information gathered during program execution in addition to information contained in source code to compute slices. Using this information, dynamic slices find program points that may affect (or may be affected by) a given variable at a given point *under a given execution*. Consequently, dynamic slicing usually produces smaller slices than static slicing. Because our purpose is to localize the possible location of faults to the smallest possible section of the program, our approach is based on dynamic slicing. In the WYSIWYT framework, checkmarks ($\surd$) on a cell $C$ are automatically propagated up $C$'s backward dynamic slice to all contributing cells. (WYSIWYT has built-in support for traversing dynamic slices, rather than static slices, when necessary.) Our approach leverages this support, as we will describe in Section 4.3.2.

Considering slicing as a fault localization technique, a slice can make use of information only about variables that have had incorrect values. But *program dicing* [38], in addition to making use of information about variables that have had incorrect values, can also make use of information about where correct values have occurred. Using dicing, faults can be further localized by "subtracting" the slices on correct variables away from the slices on the incorrect variable. Our approach is similar in concept to dicing, but is not exactly the same.

Lyle and Weiser describe the cases in which a dice on an incorrect variable not caused by an omitted statement is guaranteed to contain the fault responsible for the incorrect value in their Dicing Theorem [38]. In this theorem, the first assumption eliminates the case where an incorrect variable is misidentified as a correct variable. The second assumption removes the case where a variable is correct despite depending on an incorrect variable (e.g. when a subsequent computation happens to compensate for an earlier incorrect computation for certain inputs). The third assumption removes the case where two faults counteract each other and result in an accidently correct value. The theorem is outlined below:

**Dicing Theorem.** A dice on an incorrect variable contains a fault (except for cases where the incorrect value is caused by omission of a statement) if all of the following assumptions hold:

1. Testing has been reliable and all incorrectly computed variables have been identified.
2. If the computation of a variable, $v$, depends on the computation of another variable, $w$, then whenever $w$ has an incorrect value then $v$ does also.

3. There is exactly one fault in the program.

The Dicing Theorem [38] states that a dice is only guaranteed to contain a fault if the above three assumptions can be met. Meeting these assumptions is clearly not possible in end-user programming. As just one example, end users, like any human performing interactive testing, seem likely to make incorrect testing decisions. (We will return to this point in Sections 6 and 7.) Because of this mismatch with the audiences and environments we choose to support, our approach is different from dicing in both definition and implementation, as we describe in the remainder of this section.

## 4.3 A heuristic for estimating fault likelihood

Since computing an exact fault likelihood value for a program point in any program is not possible, a heuristic must be employed to estimate such a value based on acquired information. We have chosen to use a dicing-like procedure to formulate our approach to fault localization. Traditionally, a dice on an incorrect cell would be formed after making a binary decision about cells: either a cell is indicated as incorrect or it is not; but this does not allow for the possibility of user testing mistakes, multiple faults in a program, etc. To account for these realities, our methodology instead *estimates* the likelihood that a cell contains one or more faults that contribute to a value marked incorrect. We call this likelihood the *fault likelihood* of a cell. Let $I$ be the set of cell values marked incorrect by the program developer. The fault likelihood of a cell $C$ is an estimate of the likelihood that $C$ contains one or more faults that contribute to an incorrect value in $I$. Estimates are made according to the following five properties.

### 4.3.1 Five properties for fault localization

By piggy-backing off of the testing information base maintained by the WYSIWYT methodology, our strategy maintains five carefully chosen properties,[1] summarized below. These properties have been devised with both software engineering and human-computer interaction principles in mind and, as alluded to in Section 1, form the essence of our approach.

We will use producer-consumer terminology to keep dataflow relationships clear; that is, a *producer* of a cell $C$ contributes to $C$'s value, and a *consumer* of $C$ uses $C$'s value. In slicing terms, producers are all the cells in $C$'s backward slice, and consumers are all the cells in $C$'s forward slice. $C$ is said to *participate in a test* (or to *have a test*) if the user has marked (with a checkmark or an X-mark) $C$ or any of $C$'s consumers.

[1] In [49], we presented six properties for estimating fault likelihood, but later removed the fourth property. Properties 4 and 5 in this paper correspond to Properties 5 and 6 in [49].

**Property 1:** If cell $C$ or any of its consumers have a failed test, then $C$ will have non-zero fault likelihood.

This first property ensures that every cell that might have contributed to the computation of an incorrect value will be assigned some positive fault likelihood. This reduces the chance that the user will become frustrated searching for a fault that is not in any of the highlighted cells, which could ultimately lead to a loss of a user's trust in the system, and therefore violate the third design constraint that we placed on any fault localization strategy for form-based visual programs in Section 4.1. The property also acts as a robustness feature by ensuring that a user cannot make (possibly incorrect) testing decisions that will bring the fault likelihood of a faulty cell to zero.

**Property 2:** The fault likelihood of a cell $C$ is proportional to the number of $C$'s failed tests.

This property is based on the assumption that the more incorrect calculations a cell contributes to, the more likely it is that the cell contains a fault.

**Property 3:** The fault likelihood of $C$ is inversely proportional to the number of $C$'s successful tests.

The third property, in contrast to Property 2, assumes that the more correct calculations a cell contributes to, the less likely it is that the cell contains a fault.

**Property 4:** An X-mark on $C$ blocks the effects of any checkmarks of $C$'s consumers (forward slice) from propagating to $C$'s producers (backward slice).

This property is specifically to help narrow down where the fault is by preventing "dilution" of important clues. More specifically, producers that contribute only to incorrect values are darker, even if those incorrect values contribute to correct values further downstream. This prevents dilution of the cells' colors that lead only to X-marks. (In short, X-marks block the propagation of checkmarks.) One example of this behavior is given in Figure 3.

**Property 5:** A checkmark on $C$ blocks the effects of any X-marks on $C$'s consumers (forward slice) from propagating to $C$'s producers (backward slice), with the exception of the minimal fault likelihood property required by Property 1.

Similar to Property 4, this property uses checkmarks to prune off $C$'s producers from the highlighted area if they contribute to only correct values, even if those values eventually contribute to incorrect values. (Put another way, checkmarks block most of the propagation of X-marks.) Figure 4 depicts this behavior, and it is largely due to Property 5 that the fault's location has been narrowed down to being most likely in either `WeightedMidterm` or `Total_Score`, which are thus colored the darkest. (In fact, in Figure 4 the fault is an incorrect mathematical operation in the `WeightedMidterm` cell; instead of multiplying the value of the `Midterm` cell by 0.3, `Midterm` is multiplied by 0.4.)

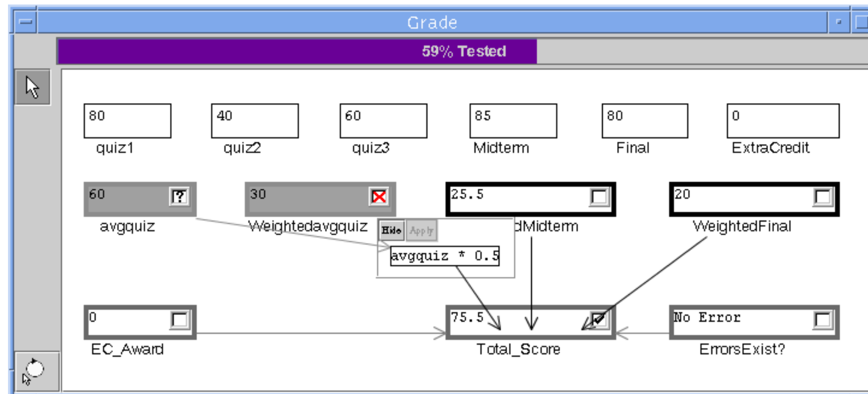


Figure 3: An illustration of Property 4's effect. In this example, the X-mark in `Weightedavgquiz` blocks any effects of the checkmark in `Total_Score` from traveling upstream past the X-mark. This has the desirable effect of coloring `Weightedavgquiz`, which is where the fault resides, as one of the two darkest cells. (It is possible for an incorrect value to occasionally or coincidentally produce a correct value downstream, and Property 4 helps to allow for this possibility. In this example, `Total_Score` has a correct value because the incorrect value of `Weightedavgquiz`—which the user has noted with an X-mark—has been "counteracted" by a second fault in the `WeightedFinal` cell, which has an incorrect value not yet noted by the user.)

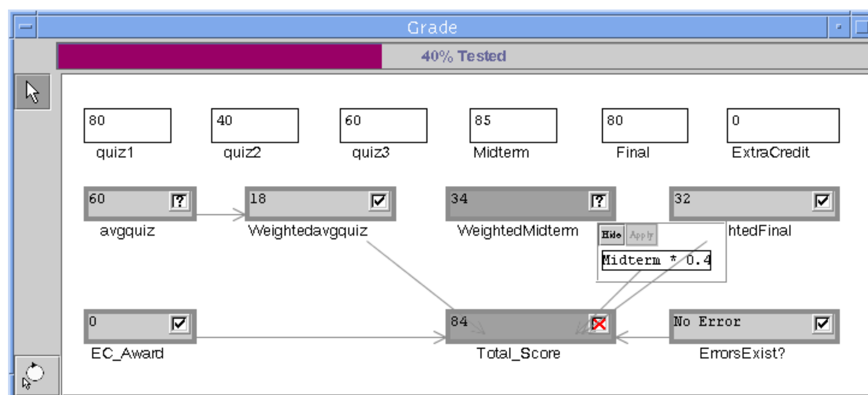


Figure 4: An illustration of Property 5's effect. In this example, the strategically placed checkmarks in the `Weightedavgquiz`, `EC_Award`, `WeightedFinal`, and `ErrorsExist?` cells block all but the minimal ("Very Low") fault likelihood from the X-mark in `Total_Score`.

| Intensity of Color | *fault likelihood(C)* |
|---|---|
| Low | 1-2 |
| Medium | 3-4 |
| High | 5-9 |
| Very High | 10+ |

Table 1: Mapping fault likelihood calculations to color intensities.

#### 4.3.2 Implementing the properties

To implement these properties, let *NumBlockedFailedTests(C)* ($NBFT$) be the number of values belonging to $C$'s consumers that are marked incorrect, but are blocked by a value marked correct along the dataflow path from $C$ to the value marked incorrect. Furthermore, let *NumReachableFailedTests(C)* ($NRFT$) be the result of subtracting $NBFT$ from the number of $C$'s consumers (i.e., the failed tests that are not blocked). Finally, let there be *NumBlockedSuccessfulTests(C)* ($NBST$) and *NumReachableSuccessfulTests(C)* ($NRST$), with definitions similar to those above.

If a cell $C$ has no failed tests, the fault likelihood of $C$ is "None". If *NumReachableFailedTests(C)* $= 0$, but *NumBlockedFailedTests(C)* $> 0$, the fault likelihood of $C$ is "Very Low". Otherwise, there are X-marks in $C$'s consumers that *reach* $C$ (are not blocked by checkmarks), and $C$'s fault likelihood is estimated using the equation below, and then mapped to a colorization using the scheme in Table 1.[2]

$$\textit{fault likelihood(C)} = max(1, 2 * NRFT - NRST)$$

As alluded to in Section 4.2, our approach leverages WYSIWYT's current dynamic slicing support for X-marks. Just as checkmarks are propagated up the marked cell's backward dynamic slice, so too are X-marks. Consequently, for each cell, our approach is able to maintain a list of all $\surd$ and X-marks affecting the cell. These lists are how we implement our five properties to calculate the fault likelihood for a cell.

The two interactions that trigger action from our fault localization strategy are (1) placing a $\surd$ or X-mark, and (2) changing a non-constant formula. Our strategy keeps track of which cells are reached by various marks so that the blocking characteristics of Properties 4 and 5 can be computed given complex, intertwined dataflow relationships. Let $p$ be the number of cell $C$'s producers, i.e., cells in $C$'s backward dynamic slice. Furthermore, let $e$ be the number of edges in the graph consisting of $C$'s backward dynamic slice, and let $m$ be the total number of marks (tests) in the program. The time complexity of placing or undoing a $\surd$ or X-mark is $O((p + e)m^2)$. Adding or modifying $C$'s non-constant formula has time complexity $O(p + em^2)$,

[2]This equation, as well as the moment when fault likelihood is mapped to a colorization scheme, has changed slightly since [49] after analyzing the results of a recent formative study of the effectiveness of our approach [55].

but in this case the edges $e$ are those of a (dataflow) multigraph because a cell can be referenced multiple times in a formula. (Details of the complexity analysis of our approach are provided in [48].)

# 5 Study

Will end users be able to benefit from the fault localization approach we have just described, and if so, how will they integrate it with their own problem-solving procedures and strategies? To explore this issue, we conducted a study investigating the following research questions:

**RQ1:** How much perceived value do end users see in the interactive, visual fault localization feedback over time?

**RQ2:** How thoroughly do end users understand the interactive, visual fault localization feedback?

**RQ3:** What debugging strategies do end users use, and what types of faults do these strategies reveal?

**RQ4:** How does fault localization feedback influence an end user's interactive debugging strategy?

**RQ5:** How do wrong testing decisions affect fault localization feedback?

To obtain the necessary qualitative information, we conducted a think-aloud study using ten end users as subjects. A think-aloud study allows subjects to verbalize the reasoning for their actions. Traditional experiments based on statistical methods provide quantitative information but do not provide the qualitative information we sought for this work. For example, a traditional experiment cannot explain user behaviors or reactions to fault localization feedback, or provide insights into the cognitive thought process of a user; rather, such experiments provide only indirect clues about the human-centric issues we sought to investigate.

## 5.1 Procedure

Our subjects were divided into two groups: a control group having only the features of WYSIWYT and a treatment group having both WYSIWYT and our integrated, visual fault localization approach. (A control group was needed for the debugging strategy comparisons of RQ3.) Each session was conducted one-on-one between an examiner and the subject. The subject was given training on thinking aloud, and a brief tutorial (described in Section 5.3) on the environment he or she would be using, followed by a practice task. Also, the environment had an on-demand explanation system via tooltips, available for all objects, reducing or eliminating the need for memorization.

After familiarizing themselves with their environment, each subject worked on the tasks detailed in Section 5.4. The data collected for each session included audio transcripts, electronic transcripts capturing all user

interactions with the system, post-session written questionnaires, and the examiner's observations. Audio transcripts captured the subjects' verbalizations as they performed the given tasks. Electronic transcripts captured user actions such as editing the values in a cell, placing a $\surd$ or X-mark in a decision box, and turning on/off arrows indicating dataflow relationships between cells. Post-session questionnaires asked about the usefulness of the WYSIWYT features in finding and fixing errors, and tested the subjects' understanding of our fault localization approach. In addition, the examiner took notes of his observations during the session.

### 5.2 Subjects

Our subjects were ten business majors with spreadsheet experience. We chose business majors because spreadsheets are a kind of form-based program commonly used for business applications. We required subjects to be experienced with spreadsheets because we did not want learning of basic spreadsheet functionality to be a variable in our study. The subjects were equally divided into two groups. We distributed subjects based on their experience with spreadsheets and their grade point average so that there was an even distribution of experienced and less experienced subjects in both groups. The information about their spreadsheet experience, programming experience, and grade point average was gathered via a background questionnaire that the participants filled out prior to the study. None of the subjects had any programming experience.

### 5.3 Tutorial

Before the study's tasks, subjects were given training on thinking aloud and a brief tutorial on the environment they would use. Since it was essential for the subjects to think aloud, the subjects were asked to do two "thinking aloud" practice problems: adding the numbers "678" and "789", and counting the number of windows in their parents' house. The tutorial about the Forms/3 environment and WYSIWYT was designed such that the subjects received sufficient practice editing, testing, and debugging a cell. The tutorial for both groups was the same except that the feedback of our fault localization approach was included in the treatment group's tutorial and used to debug a cell. To counterbalance this material, the control group did the same debugging activity without any fault localization feedback.

For the tutorial, the subjects performed tasks on their own machines with guidance at each step. Subjects were free to ask questions or seek clarifications during the tutorial. The tutorial ended when the subjects were judged to have learned the features of Forms/3. At the end of the tutorial, the subjects were given two minutes to explore the program they were working with during the tutorial to allow them to work further with the features taught in the tutorial. As a final tutorial task, to prepare the subjects for the study's tasks, subjects were given five minutes to practice debugging a simple program.

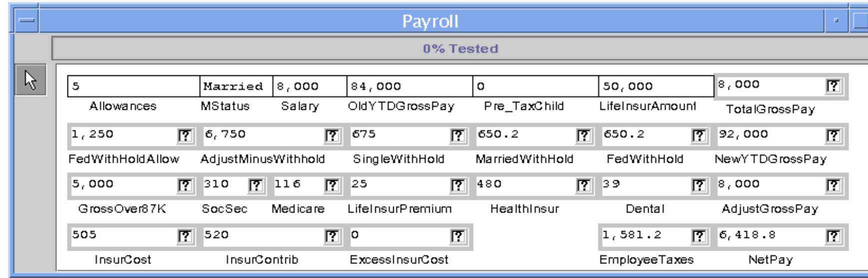


Figure 5: The `Payroll` program.

## 5.4 Tasks

Allwood classified faults in spreadsheets as mechanical, logical, and omission faults [1], and this scheme is also used in Panko's work [44]. Under Allwood's categorization, mechanical faults include simple typographical errors or wrong cell references in the cell formulas. Mistakes in reasoning were classified as logical faults. Logical faults are more difficult than mechanical faults to detect and correct, and omission faults are the most difficult [1]. An omission fault is information that has never been entered into the formula.

Drawing from this research, we seeded five faults of varying difficulty into each of two Forms/3 programs. One of these programs is the `Grades` program from Figure 1, which computes the total score for a course given input for three quizzes, extra credit, a midterm, and a final exam. There is also a cell that indicates when an input cell is outside the valid range. `Grades` had three mechanical faults, one logical fault, and one omission fault. This program was designed to be the "easier" of our two tasks based on its size, the complexity of its formulas, and our choice of seeded faults.

The other program, `Payroll`, is presented in Figure 5. Into `Payroll`, we seeded one mechanical fault, three logical faults, and one omission fault. This program was much larger, had more levels of data flow, and had a greater number of cells with non-constant formulas in relation to input cells when compared to the `Grades` program.[3]

Subjects were given these two programs (tasks) in varying order, with instructions to test and correct any errors found in the programs. For each task, the subjects were provided the unverified program and a description of the program's functionality. Furthermore, the subjects were provided a single example of the expected output values given specified inputs for the `Grades` task, and two such examples for the `Payroll` task. Subjects had a time limit of 15 minutes to complete the `Grades` task and 30 minutes to complete the `Payroll` task.

[3] The starting formulas for both tasks can be found in [46].

# 6 Results

## 6.1 RQ1: How much perceived value do end users see in the feedback over time?

Blackwell's model of attention investment [8] is one model of user problem-solving behavior predicting that users will not want to enter an X-mark unless the benefits of doing so are clear to them. The model considers the costs, benefits, and risks users weigh in deciding how to complete a task. For example, if the ultimate goal is to forecast a budget using a spreadsheet, then using a relatively unknown feature such as an X-mark has cost, benefit, and risk. The costs are figuring out when and where to place the X-mark and thinking about the resulting feedback. The benefit of finding faults may not be clear after only one X-mark; in fact, the user may have to expend even more costs (place more X-marks) for benefits to become clear. The risks are that going down this path will be a waste of time, or worse, will mislead the user into looking for faults in the correct formulas instead of in the incorrect ones.

First, we consider whether users, having briefly seen X-marks in the tutorial, were willing to place even one X-mark to help their debugging efforts. The answer was that they were: four of the five treatment subjects placed at least one X-mark, especially when they needed assistance debugging a failure (discussed further in Section 6.4). The subject who did not place an X-mark (treatment subject TS3) explained during a post-session interview that she had forgotten about them, and wished she had used them:

**TS3:** *I wish I could redo the problem with the X-mark. If I would have done that, it would have been a lot more easier.*

In our interactive fault localization system, the first interaction about a failure (X-mark) leads to feedback, and this feedback may or may not provide enough benefits to the user to lead him or her to place a second X-mark. In general, a difference in any *interactive* fault localization approach from traditional approaches is that the accuracy of feedback about fault locations must be considered at every step of the way, especially in early steps, not just at the end of some long batch of tests. As the attention investment model explains, if the early feedback is not seen as providing information that is truly of practical help, there may never be any more interactions with the system! This was exactly the case for subject TS5, who placed only one X-mark in the `Grades` task, and explained after the session:

**TS5:** *To me, putting an X-mark just means I have to go back to do more work.*

In his case, the X-mark he placed was in a cell whose only contributors were input cells; consequently, because our fault localization approach does not tint input cells (which have constant formulas rather than non-constant formulas), the only cell tinted was the cell in which he just placed the X-mark. Since this

feedback did not add to the information he already had, it is not surprising that he found no benefit from placing an X-mark. This indicates the importance of the feedback (reward), even in the early stages of use; if the reward is not deemed sufficient for further attention, a user may not pursue further use.

However, the other three treatment subjects who placed an X-mark went on to place a second X-mark later in the session. Further, after placing this second X-mark, all three subjects then went on to place a third X-mark. The successes that rewarded these subjects (detailed in Section 6.3) outweighed their perceived costs of testing and marking failures with X-marks, indicating that the rewards of placing X-marks were sufficiently conveyed to these users.

## 6.2 RQ2: How thoroughly do end users understand the interactive feedback?

To what extent did the subjects understand the message the interactive feedback was trying to communicate? We investigated two levels of understanding: the deeper level being the ability to predict feedback under various circumstances, and the more shallow level of being able to interpret feedback received.

To investigate these two levels of understanding, the post-session questionnaire for our treatment subjects had 11 questions, approximately evenly divided between the two levels, regarding the effects of X-marks on the interior colorings of a cell. The subjects' ability to *predict* behavior, as measured by six questions, was mixed. Again using producer-consumer terminology, all subjects were able to correctly predict the impacts on producer cells of placing a single X-mark (two questions). About half the subjects were able to predict the impacts on consumer cells of placing a single X-mark (one question) and to predict the impacts when multiple X- and checkmarks were involved (three questions). However, the ability to *interpret* behaviors was uniformly good: all four of the subjects who actually used X-marks during the session were able to explain the meanings of the colorings and marks, and to say what those meanings implied about faults (five questions). For example, some responses to questions about what it means when the interior of cells get darker or get lighter were:

**TS1:** *If the color becomes lighter, the cells have less of a chance of to be wrong.*

**TS2:** *The chance of an error in the darker cells is greater than in the lighter cells.*

**TS4 (referring to a darker cell):** *Higher chance of error in that cell.*

These post-session questionnaire results are corroborated by the actions of the users themselves, as we will discuss in the next two sections.

| | Ad hoc | Dataflow | | Total |
|---|---|---|---|---|
| | | dataflow total | using X-marks | |
| *Grades:* | | | | |
| Control | 13/20 (65%) | 6/6 (100%) | n/a | 19/26 (73%) |
| Treatment | 13/16 (81%) | 9/10 (90%) | 5/5 (100%) | 22/26 (85%) |
| Total | 26/36 (72%) | 15/16 (94%) | | 41/52 (79%) |
| *Payroll:* | | | | |
| Control | 6/17 (35%) | 3/6 (50%) | n/a | 9/23 (39%) |
| Treatment | 9/21 (43%) | 6/12 (50%) | 3/5 (60%) | 15/33 (45%) |
| Total | 15/38 (39%) | 9/18 (50%) | | 24/56 (43%) |

Table 2: The success rates of identifying a fault contributing to an observed failure (faults identified / failures observed), for each debugging strategy. For each row and column except "Total", a numerator of 25 is the maximum possible. (The maximum possible is 25 because each task had five faults, and the control and treatment groups who worked on these tasks were each composed of five subjects.)

### 6.3 RQ3: What debugging strategies do end users use, and what types of faults do these strategies reveal?

Because this work is about fault *localization*, we focus on users' abilities to *identify* the location of faults, as defined by either an explicit verbal statement or by the fact that they edited the cell's formula. (Once identified, corrections usually followed; 60 of the 69 faults were corrected once identified.)

Once a failure was spotted, users exhibited two kinds of debugging strategies to find the fault causing the failure: an ad hoc strategy, in which they examined cell formulas randomly in no particular order, and a dataflow strategy, in which they followed the failure's dependencies back through cell references until they found the fault. A dataflow strategy can be accomplished through mental effort alone, but subjects rarely did this: mostly they used either arrows, the fault localization feedback, or a combination of both.

Table 2 enumerates the subjects' strategy choices and corresponding success rates. Comparing the first two columns' percentages column-wise shows that, for both subject groups, dataflow debugging tended to be more successful than ad hoc. Within dataflow, the treatment subjects' success rates with X-marks exceeded the dataflow total success rates. A row-wise comparison of the denominators in the table also shows that treatment subjects tended to move to dataflow strategies nearly twice as frequently as the control subjects. (The table's numerators cannot be compared column-wise or row-wise as raw values, because subjects usually started with ad hoc and then sometimes switched to dataflow for hard-to-locate faults.)

These differences in strategy choices lead to the following question: In what situations did the strategies matter?

*Easy faults:* The subjects' strategy choices did not matter with the easiest faults. The easiest are mechanical faults, according to Allwood [1], and were usually found regardless of the strategy used. Over all tasks and all subjects, 35 of the 40 mechanical faults were identified.

*Local faults:* Strategy also did not matter much with the "local" faults (those in which the failed value spotted by the subject was in the same cell as the faulty formula). This is often the case in smaller programs, where there are fewer cells to reference and the likelihood of a local fault is greater, and probably contributed to both groups' greater success in the `Grades` task.

*Non-local faults:* Strategy mattered a great deal for the non-local faults. Over all of the subjects and tasks, 16 non-local faults were identified—all using dataflow. *Not a single non-local fault was identified using the ad hoc strategy.* In fact, for seven of these non-local fault identifications (by six different subjects), the subjects began their search for the fault using an ad hoc strategy and, when unable to succeed, switched to a dataflow strategy, with which they succeeded in finding the fault.

Our fault localization strategy augments the dataflow strategy, which is especially illustrated by treatment subjects TS4 and TS5. Both subjects found all faults in the smaller `Grades` task. Both subjects also found the mechanical fault and one of the logical faults in the larger `Payroll` task in short order. But then, they both got stuck on where to go next. At this critical juncture, TS4 decided to place an X-mark (carried out with a right-click) on a failure. Once he saw the feedback, he rapidly progressed through the rest of the task, placing five X-marks and correcting the final three faults in only seven minutes. The transcripts show that the initial X-mark, which initiated the (dataflow-oriented) fault localization feedback, was a key turning point for him:

**TS4 (thinking aloud):** *Right click that 'cause I know it's incorrect, highlights everything that's possible error . . .* `EmployeeTaxes` *is also incorrect. My* `NetPay` *is incorrect. Adjusted gross pay (*`AdjustGrossPay`*) is incorrect, so click those wrong.*

Whereas TS4 made the defining decision to use the X-mark, TS5 did not. TS5's pattern of looking at cells gradually became ad hoc. He started randomly looking at formulas. He made decisions about the values in various cells and eventually happened upon a local fault, bringing his total to three. He never searched the dataflow path any further than one cell. He said "*I'm getting confused here*" numerous times, but did not change his approach.

## 6.4 RQ4: How does this feedback influence an interactive debugging strategy?

We had initially expected that treatment subjects would always place X-marks whenever they observed a failure and use the subsequent visual feedback to guide their debugging, but this was not the case. Instead, they seemed to view the X-marks as a device to be called upon only when they were in need of assistance. For example, only late in the session, when treatment subject TS1 got stuck debugging the failures, did he turn to the fault localization strategy:

**TS1 (thinking aloud):** *I don't know how to check the kind of error it is. I'll mark it wrong and see what happens.*

When subjects did place an X-mark, the visual feedback often had an immediate impact: regardless of what their previous strategy had been, as soon as the feedback appeared, the subjects switched to a dataflow strategy by limiting their search to those cells with estimated fault likelihood and ignoring cells with no assigned fault likelihood.

**TS1 (thinking aloud):** *I'm going to right-click on the* `Total_Score`*. See that the weighted average, the weighted quiz (*`Weightedavgquiz`*), the* `WeightedMidterm`*, and the* `WeightedFinal`*, and the error box (*`ErrorsExist?`*) all turn pink.*

The fault localization device beckons the user toward a dataflow strategy, but it has attributes dataflow arrows do not have. First, it produces a smaller search space than the dataflow arrows, because it highlights only the producers that actually *did* contribute to a failure (a combination of dynamic slices), rather than including the producers that could contribute to failures in other circumstances (a combination of static slices). Second, it prioritizes the order in which the users should consider the cells, so that the faulty ones are likely to be considered earliest. The above shows that TS1 took advantage of the reduction in search space brought about by the tinting of the producers of a cell with a failure. But did the subjects also take advantage of this prioritization, indicated by some cells being darker than others?

Our electronic transcripts indicate that the answer to this question is yes. When the subjects searched cell formulas for a fault after placing an X-mark, 77% of these searches initially began at the cell with the darkest interior shading. As an example, here is a continuation of the above quote from TS1 after placing an X-mark:

**TS1 (thinking aloud):** *See that the weighted average, the weighted quiz (*`Weightedavgquiz`*), the* `Weighted-Midterm`*, and the* `WeightedFinal`*, and the error box (*`ErrorsExist?`*) all turn pink. The* `Total_Score` *box is darker though.*

Another example can be found in a post-session questionnaire response from TS5:

**TS5:** *The problem has been narrowed to the darker cells.*

When the fault was not in the darkest cell, subjects' searches would gradually progress to the next darkest cell and so on. Some subjects realized that the colorings' differentiations could be enhanced if they made further testing decisions by placing $\surd$ and X-marks, carried out by left- or right-clicking a cell's decision box.

**TS4 (thinking aloud):** *Right click that 'cause I know it's incorrect, highlights everything that's possible errors. Now, I know my* `TotalGrossPay` *is correct. I'll left click that one and simplify it.*

From the results of this and the previous sections, it is clear that fault localization's ability to draw the user into a suitable strategy (dataflow) was important, particularly when subjects had not figured out a strategy that would help them succeed better than ad hoc approaches. Further, it is clear that subjects were influenced by the feedback's prioritization information when more than one color was present—in that they looked first to the darkest cells, and then to the next darkest, and so on—and that their doing so increased success.

## 6.5 RQ5: How do wrong testing decisions affect fault localization feedback?

Being human, the end-user subjects in our study made some mistakes in their testing decisions. Here we consider the types of mistakes they made, and the impact of these mistakes on the users' successful use of the fault localization feedback. (Because the control subjects did not have fault localization feedback, we consider only the treatment subjects.)

In total, the five treatment subjects placed 241 checkmarks, of which 11 (4.56%) were wrong—that is, the user pronounced a value correct when in fact it was incorrect. Surprisingly, however, none of the 15 X-marks placed by subjects were incorrect.

A possible reason for this difference may be a perceived seriousness of contradicting a computer's calculations, meaning subjects were only willing to place X-marks when they were really sure their decision was correct. (This is consistent with observations in an earlier small study [7] of a subject exhibiting great faith in the computer's output.) For example, at one point, subject TS1 placed an X-mark in a cell, then reconsidered the mark because he was unsure the X-mark was really warranted.

**TS1 (thinking aloud):** *So, I'll right click on that one. I'm not sure if this is right. Eh, I'll leave it as a question mark.*

In contrast, checkmarks were often placed even if the user was unsure whether the marks were warranted. Our verbal transcripts include ten different statements by treatment subjects with this sentiment. For example, consider the following quotes from two treatment subjects:

**TS1 (thinking aloud):** *I'll go ahead and left click the* `LifeInsurPremium` *box because I think that one's right for now.*

**TS3 (thinking aloud):** *I think these are right, (so) check that.*

What impact did the wrong checkmarks have on fault localization? Four of the 11 wrong checkmarks were placed with a combination of X-marks, resulting in incorrect fault localization feedback. All four of these particular checkmarks, placed by three different subjects, adversely affected the subjects' debugging efforts.

For example, during the `Grades` task, TS1 placed an incorrect checkmark in the (faulty) `WeightedMidterm` cell. He later noticed that the `Total_Score` cell, although its formula was correct, had an incorrect value (due to the fault in `WeightedMidterm`). Unable to detect the source of this failure, he turned to the fault localization strategy and placed an X-mark in the `Total_Score` cell:

**TS1 (thinking aloud):** *The* `Total_Score` *box is darker though. And it says the error likelihood is low, while these other boxes that are a little lighter say the error likelihood is very low. Ok, so, I'm not sure if that tells me anything.*

The subject knew that `Total_Score` was correct. Figure 6 illustrates that had it not been for the wrong checkmark, the faulty `WeightedMidterm` cell would have been one of the two darkest cells in the program. Instead, the wrongly placed checkmark caused `WeightedMidterm` to be colored the same as its correct siblings, thus providing the subject with no insightful fault localization feedback. (The subject eventually corrected the fault after a search of over six minutes.)

Subject TS2, faced with a similar scenario as in Figure 6, was overcome with frustration and confusion:

**TS2 (thinking aloud):** *All right...so, I'm doing something wrong here. (long pause) I can't figure out what I'm doing wrong.*

TS2's confusion resulted in nearly seven minutes of inactivity. He eventually located and corrected the fault, but remained flustered for the duration of the session.

As this evidence makes clear, it would not be realistic to ignore the fact that end users will provide some wrong information. In our study, even though fewer than 5% of the checkmarks placed by the subjects were wrong, these marks affected 60% (three out of five) of the treatment subjects' success rates! Given the presence of mistakes, robustness features may be necessary to allow success even in the presence of mistakes. Toward this end, recall from Section 4 that our fault localization strategy colors every cell in the dataflow chain contributing to a failure—even the cells the user may have previously checked off. Clearly, however, this attempt at robustness was not enough to completely counteract the impacts of mistakes. Alternative

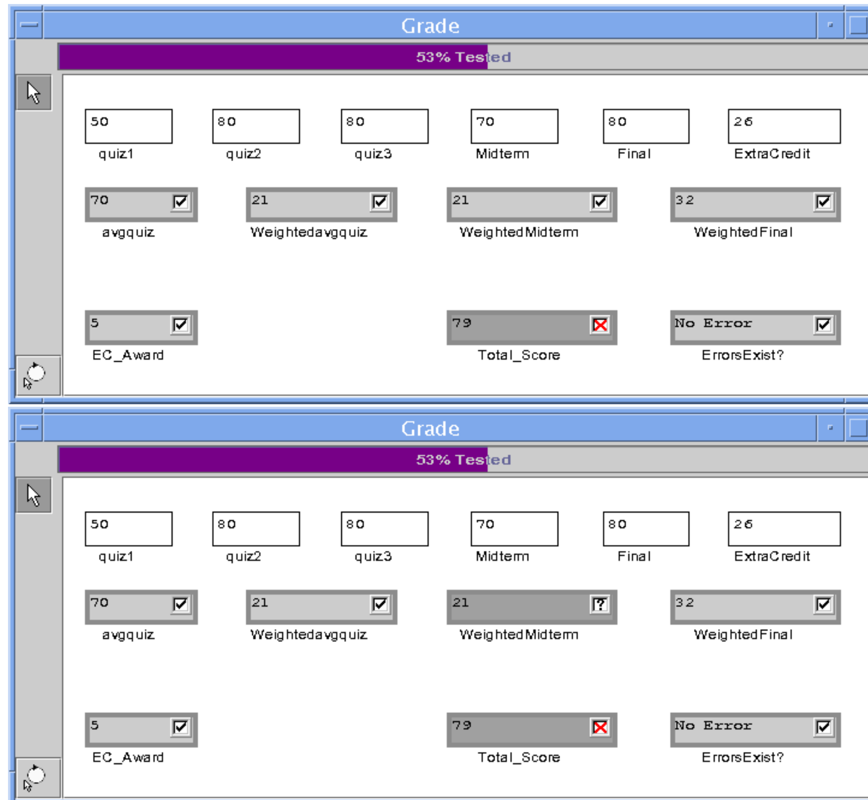


Figure 6: (Top) The `Grades` task, with an incorrect checkmark in `WeightedMidterm`, as seen by subject TS1. `Total_Score` is the darkest, and the other six all are the same shade. (Bottom) What TS1 would have seen without the wrong checkmark: `WeightedMidterm` would be as dark as `Total_Score`.

strategies whose build-up of historical information can overcome some number of errors [55] are another possibility.

# 7 Discussion

Previous fault localization research has focused primarily on techniques to aid *professional programmers* performing *batch* testing. In contrast, our study focuses on supporting *end-user programmers* with an *interactive* fault localization strategy. In generalizing our experiences in end-user debugging, there are many insights that future work in this area may need to consider.

## 7.1 When was fault localization used?

We expected that subjects would place X-marks on every cell with an incorrect output value (failure). We further expected that subject verbalizations and electronic transcripts would show that our interactive, visual

feedback effectively guided the debugging efforts of subjects. While our feedback proved helpful to subjects when they actually triggered it, in most cases, subjects did not initially invoke our fault localization strategy for assistance. Instead, they treated fault localization as a resource to be called upon only when they had exhausted their own debugging abilities.

The reason for this behavior might be tied to attention investment [8]. According to this model, users manage their attention (time) by attempting to maximize their benefit-cost return in terms of the time spent to get their work done. Applied to debugging, this suggests that as long as users are progressing at a good rate at finding the faults (the benefit), users may see no reason to expend time (cost) on figuring out which cells to mark correct or incorrect. However, once the easiest faults have been found and the time (cost) to find more faults starts to increase, the extra time to mark cells may be hypothesized as being worthwhile, potentially saving time on the "hard-to-find" faults. Users would then be likely to evaluate this hypothesis by whether their change in strategy was helpful (increased benefits) in the form of more faults found.

## 7.2 When was fault localization helpful?

Many of the faults that contributed to failures observed by subjects were either "local" to the failure or would be classified by Allwood as "easy" [1]. In both cases, subjects were usually able to locate the fault relatively easily, and without the assistance of our fault localization strategy (or even a dataflow-based debugging strategy). Hence, fault localization support did not appear to be especially useful to end users when searching for easy and/or local faults, a result we did not anticipate.

However, fault localization proved especially useful for subjects searching for non-local faults. Some subjects realized without help that a dataflow strategy was needed, but some did not. While dataflow-based debugging strategies may seem intuitively superior in the eyes of trained computer scientists, our study indicates that such strategies may not come naturally to end-user programmers. One key way our fault localization approach helped was to lead users into a suitable strategy. (Recall from Section 6.3 that not a single non-local fault was identified using an ad hoc strategy; all 16 identified non-local faults were localized using a dataflow-based debugging strategy.) Once subjects were engaged in a suitable strategy, fault localization helped further by prioritizing the order they should follow the strategy.

## 7.3 Mistakes in end-user debugging

Our study brings a unique approach to fault localization by considering *interactive* debugging with respect to end-user programmers. In exploring this issue, we found that end users are prone to making mistakes during interactive debugging. Moreover, these mistakes had a big impact on the success of the majority of

our treatment subjects. This brings up a variety of issues that future work on end-user debugging may need to consider.

Recall from Section 4 that we employed a dicing-like heuristic as the cornerstone of our fault localization approach. We hypothesized that end users might make mistakes during interactive debugging, and such mistakes would violate the first assumption of the Dicing Theorem [38] that testing has been reliable. (In fact, the correctness of testing decisions is a common assumption in software engineering research.) This hypothesis was validated through our study, as 4.56% of the WYSIWYT checkmarks placed by our subjects were incorrectly assigned. Therefore, fault localization strategies explored in previous research, such as program dicing, may be inherently unsuitable for end-user programming environments.

In terms of fault localization design, we found that end users were prone to making incorrect testing decisions regarding perceived correct values (i.e., erroneously-placed checkmarks in WYSIWYT), but never pronounced values incorrect when they were in fact correct (i.e., erroneously-placed X-marks). It would clearly not be responsible to claim that end users will never place incorrect X-marks within the WYSIWYT methodology. However, our study does indicate the end-user programmers are much more likely to make mistakes regarding values perceived to be correct (checkmarks) than values perceived to be incorrect (X-marks). One lesson developers might take away from this result is to place greater trust in the "X-mark-equivalent" decisions made by users, and be more wary when considering the impact of a checkmark on fault likelihood calculations. Fault localization strategies may also benefit from including features to enhance robustness in the face of a few mistakes.

## 7.4 The importance of early feedback

In our research into fault localization for end users, we strove for a strategy that would provide useful feedback at all stages of the debugging process, from beginning to end. However, our study helped us to realize that we underestimated the importance of any strategy's *early* feedback when targeting end users. Our study made clear that if a strategy's early feedback is not seen to be useful, users may not give the strategy a chance to produce better feedback later.

This may be the most challenging result of our study from the perspective of future fault localization strategy developers. It is not surprising that previous fault localization research has not focused on strategies providing useful early feedback—after all, early feedback is of little consequence to professional programmers performing batch testing of test suites. Yet this early feedback may be paramount to the success of an interactive fault localization strategy in an end-user programming environment. The lesson for developers

is that the fault localization strategies best suited for end users may be those that provide quality early feedback, even if sacrificing the quality of later feedback is necessary.[4]

# 8 Threats to Validity

Every study has threats to the validity of its results. This section discusses potential threats to the validity of our study and, where possible, how we attempted to mitigate the impact of these threats on our results.

Threats to internal validity are other factors that may be responsible for our results. It is possible that the specific faults seeded in our two programs are responsible for our results. To mitigate this factor, as described in Section 5.4, we seeded faults according to Allwood's classification scheme [1] to ensure that different types of faults were seeded in our tasks.

Threats to construct validity question whether the results of a study are based on appropriate information. Because of the qualitative information necessary to answer our research questions, many of our results come from verbalizations of our subjects. However, one limitation of using subject verbalizations is the possibility that our subjects did not vocalize some of their thought processes during the completion of the study's tasks. If this was the case, it is possible that we were deprived of information that could have changed the conclusions of our study. We attempted to mitigate this possibility by politely requesting that subjects "think aloud" when they displayed periods of silence. It is also possible that the one-on-one nature of the study between the subject and the examiner intimidated the subjects or caused them to try to give answers they thought the examiner wanted to hear. To diminish concern over this issue, the only questioning of subjects was written, and this occurred only after the completion of a task. However, these two limitations are inherent in any think-aloud study.

Threats to external validity limit the extent to which results can be generalized. In considering this matter, program representativeness is an issue facing our study. If the programs used in our study did not mimic those that real end users create, our results may not generalize. Also, to better control study conditions and ensure that subjects could complete the tasks in the alloted time, our programs were not large, and therefore may not necessarily be representative of real end-user programs. In future work, although it may sacrifice some internal validity, a solution to this issue may be to replicate this study using programs obtained from real end users with real faults. An alternative solution may be to gather additional empirical evidence through future studies using a greater range and number of programs. Finally, the fault seeding process was chosen to help control the threats to internal validity, but this came at a cost of some external

[4]A recent formative experiment [55] examines the quality of the early and later feedback provided by the strategy presented in this paper, as well as the feedback of two separate fault localization strategies.

validity, because the faults and fault patterns of real end users may differ from those introduced into our study tasks.

# 9 Fault Localization in Other Paradigms

The design of a fault localization strategy can be broken down into three components: the "information base" maintained by the strategy, how that information base is leveraged by a heuristic to estimate the fault likelihood of a given program point, and how that fault likelihood is communicated to a user. Given these components, we believe that it is feasible to extend our strategy to other programming paradigms. Because the visualization mechanisms of communicating fault likelihood calculations are highly dependent on the targeted paradigm, we discuss some issues that may arise in extending our strategy's information base and heuristic to other paradigms, beginning with the implications stemming from our heuristic.

## 9.1 Heuristic implications

As mentioned in Sections 1 and 4.3.1, we based our heuristic around five properties, which were carefully chosen with human-centric principles in mind as much as software engineering considerations. These properties reflect how the information base is manipulated (e.g., tracking the number of blocked failed tests) just as much as the mathematical calculations that map testing information to fault likelihood values.

A happy characteristic of these properties is their generality. Although our properties were stated using cells and their values as examples of variables and cell formulas as examples of program points, the properties could be applied to any paradigm's equivalent variable and program point structures. For example, in Prograph [22], each node with its incoming wires is a program point corresponding to a cell formula, and each incoming or outgoing port on a dataflow node is a variable corresponding to a cell value. It follows that the mathematical calculations that map testing information to fault likelihood values can also be adjusted as needed in other programming environments.

This generality means that the only assumption made by our properties is that the system has knowledge of which tests were dynamically exercised by each program point. Thus, we turn to the availability of systems and methodologies to track this information next.

## 9.2 Information base

The information base of our strategy is a record of tests affecting each variable. To obtain this testing information, we coupled our fault localization approach to the WYSIWYT testing methodology; therefore, because of the generality of the properties, any paradigm capable of supporting WYSIWYT is capable of supporting our fault localization strategy.

Although WYSIWYT was designed for form-based visual programs, it is generalizable to other programming paradigms as well. For example, there has been Prograph-based work to adapt the methodology to the dataflow paradigm [32]. WYSIWYT has also been successfully extended to the screen transition paradigm [10], which has a strong relationship with rule-based programming and also to more traditional state transition diagrams. However, even without WYSIWYT, any other form of maintaining the same testing information would also provide the necessary information base.

## 10 Conclusions

This paper presents an entirely visual, interactive approach to fault localization for end-user programmers. Our approach estimates the likelihood that each program point contains a fault, and interactively updates this fault likelihood information as future testing decisions are made by a user. The results of our study suggest that our visual fault localization approach can substantially improve the debugging efforts of end users because it guides users into more effective dataflow-based debugging strategies and eases the localization of difficult-to-detect faults. We hope that the results of our study will help developers of future fault localization strategies and future programming environments better address the unique needs of end-user programmers.

## Acknowledgments

We thank the members of the Visual Programming Research Group at Oregon State University for their feedback and help. This work was supported in part by NSF under ITR-0082265 and in part by the EUSES Consortium via NSF grant ITR-0325273.

## References

[1] C. Allwood. Error detection processes in statistical problem solving. *Cognitive Science*, 8(4):413–437, 1984.

[2] A. Ambler. The Formulate visual programming language. *Dr. Dobb's Journal*, pages 21–28, August 1999.

[3] A. Ambler, M. Burnett, and B. Zimmerman. Operational versus definitional: A perspective on programming paradigms. *Computer*, 25(9):28–43, September 1992.

[4] ANSI/IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE, New York, 1983.

[5] J. Atwood, M. Burnett, R. Walpole, E. Wilcox, and S. Yang. Steering programs via time travel. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 4–11, Boulder, Colorado, September 3–6, 1996.

[6] Y. Ayalew and R. Mittermeir. Spreadsheet debugging. In *Proceedings of the European Spreadsheet Risks Interest Group*, Dublin, Ireland, July 24–25, 2003.

[7] L. Beckwith, M. Burnett, and C. Cook. Reasoning about many-to-many requirement relationships in spreadsheets. In *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments*, pages 149–157, Arlington, VA, September 3–6, 2002.

[8] A. Blackwell. First steps in programming: A rationale for attention investment models. In *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments*, pages 2–10, Arlington, VA, September 3–6, 2002.

[9] B.W. Boehm, C. Abts, A.W. Brown, and S. Chulani. *Software Cost Estimation with COCOMO II.* Prentice Hall PTR, Upper Saddle River, NJ, 2000.

[10] D. Brown, M. Burnett, G. Rothermel, H. Fujita, and F. Negoro. Generalizing WYSIWYT visual testing to screen transition languages. In *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments*, pages 203–210, Auckland, New Zealand, October 28–31, 2003.

[11] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, March 2001.

[12] M. Burnett, S. Chekka, and R. Pandey. FAR: An end-user language to support cottage e-services. In *Proceedings of the IEEE Symposium on Human-Centric Languages*, pages 195–202, Stresa, Italy, September 2001.

[13] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the* $25^{th}$ *International Conference on Software Engineering*, pages 93–103, Portland, OR, May 3–10, 2003.

[14] M. Burnett and H. Gottfried. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction*, 5(1):1–33, March 1998.

[15] M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang. Toward visual programming languages for steering in scientific visualization: A taxonomy. *IEEE Computer Science and Engineering*, 1(4), 1994.

[16] M. Burnett, G. Rothermel, and C. Cook. An integrated software engineering approach for end-user programmers. In H. Lieberman, F. Paterno, and V. Wulf, editors, *End User Development*. Kluwer Academic Publishers, 2004 (to appear).

[17] M. Burnett, A. Sheretov, and G. Rothermel. Scaling up a 'What You See Is What You Test' methodology to spreadsheet grids. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 30–37, Tokyo, Japan, September 13–16, 1999.

[18] D.A. Carr. End-user programmers need improved development support. In *CHI 2003 Workshop on Perspectives in End-User Development*, pages 16–18, April 2003.

[19] E. Chi, J. Riedl, P. Barry, and J. Konstan. Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applications*, July/August 1998.

[20] C. Cook, M. Burnett, and D. Boom. A bug's eye view of immediate visual feedback in direct-manipulation programming systems. In *Proceedings of Empirical Studies of Programmers: Seventh Workshop*, pages 20–41, Alexandria, VA, October 1997.

[21] C. Corritore, B. Kracher, and S. Wiedenbeck. Trust in the online environment. In *HCI International*, volume 1, pages 1548–1552, New Orleans, LA, August 2001.

[22] P.T. Cox, F.R. Giles, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *IEEE Workshop on Visual Languages*, pages 150–156, Rome, Italy, October 4–6, 1989.

[23] D. Cullen. Excel snafu costs firm $24m. *The Register*, June 19, 2003. http://www.the-register.co.uk/content/67/31298.html.

[24] J.S. Davis. Tools for spreadsheet auditing. *International Journal on Human-Computer Studies*, 45:429–442, 1996.

[25] S. Eick. Maintenance of large systems. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 315–328. MIT Press, Cambridge, MA, 1998.

[26] M. Fisher, M. Cao, G. Rothermel, C.R. Cook, and M.M. Burnett. Automated test case generation for spreadsheets. In *Proceedings of the* $24^{th}$ *International Conference on Software Engineering*, pages 141–151, Orlando, Florida, May 19–25, 2002.

[27] M. Fisher II, D. Jin, G. Rothermel, and M. Burnett. Test reuse in the spreadsheet paradigm. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, November 2002.

[28] M. Heath, A. Malony, and D. Rover. Visualization for parallel performance evaluation and optimization. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 347–365. MIT Press, Cambridge, MA, 1998.

[29] N. Heger, A. Cypher, and D. Smith. Cocoa at the Visual Programming Challenge 1997. *Journal of Visual Languages and Computing*, 9(2):151–168, April 1998.

[30] T. Igarashi, J.D. Mackinlay, B.-W. Chang, and P.T. Zellweger. Fluid visualization of spreadsheet structures. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 118–125, 1998.

[31] J.A. Jones, M.J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the* $24^{th}$ *International Conference on Software Engineering*, pages 467–477, Orlando, Florida, May 19–25, 2002.

[32] M. Karam and T. Smedley. A testing methodology for a dataflow based visual programming language. In *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments*, pages 280–287, Stresa, Italy, September 5–7, 2001.

[33] D. Kimelman, B. Rosenburg, and T. Roth. Visualization of dynamics in real world software systems. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 293–314. MIT Press, Cambridge, MA, 1998.

[34] A.J. Ko and B.A. Myers. Designing the Whyline: A debugging interface for asking questions about program failures. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, page (to appear), Vienna, Austria, April 24–29, 2004.

[35] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, November 1990.

[36] V. Krishna, C. Cook, D. Keller, J. Cantrell, C. Wallace, M. Burnett, and G. Rothermel. Incorporating incremental validation and impact analysis into spreadsheet maintenance: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 72–81, Florence, Italy, November 2001.

[37] H. Lieberman and C. Fry. ZStep 95: A reversible, animated source code stepper. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 277–292. MIT Press, Cambridge, MA, 1998.

[38] J.R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *Proceedings of the* $2^{nd}$ *International Conference on Computers and Applications*, pages 877–883, 1987.

[39] R.C. Miller and B.A. Myers. Outlier finding: Focusing user attention on possible errors. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 81–90, November 2001.

[40] B. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 243–249, May 1991.

[41] D. Notkin, R. Ellison, G. Kaiser, E. Kant, A. Habermann, V. Ambriola, and C. Montanegero. Special issue on the Gandalf project. *Journal on Systems and Software*, 5(2), May 1985.

[42] H. Pan and E. Spafford. Toward automatic localization of software faults. In *Proceedings of the* $10^{th}$ *Pacific Northwest Software Quality Conference*, October 1992.

[43] R. Panko. Finding spreadsheet errors: Most spreadsheet errors have design flaws that may lead to long-term miscalculation. *Information Week*, page 100, May 1995.

[44] R. Panko. What we know about spreadsheet errors. *Journal on End User Computing*, pages 15–21, Spring 1998.

[45] S. Peyton Jones, A. Blackwell, and M. Burnett. A user-centered approach to functions in Excel. In *Proceedings of the ACM International Conference on Functional Programming*, pages 165–176, Uppsala, Sweden, August 25–29, 2003.

[46] S. Prabhakararao and C. Cook. Interactive fault localization for end-user programmers: A think aloud study. Technical Report 04-60-03, Oregon State University, Corvallis, OR, January 2004.

[47] S. Prabhakararao, 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 *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments*, pages 15–22, Auckland, New Zealand, October 28–31, 2003.

[48] J. Reichwein and M.M. Burnett. An integrated methodology for spreadsheet testing and debugging. Technical Report 04-60-02, Oregon State University, Corvallis, OR, January 2004.

[49] J. Reichwein, G. Rothermel, and M. Burnett. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. In *Proceedings of the* $2^{nd}$ *Conference on Domain Specific Languages*, pages 25–38, October 1999.

[50] S. Reiss. Graphical program development with PECAN program development systems. In *Proceedings of the Symposium on Practical Software Development Environments*, April 1984.

[51] S. Reiss. Visualization for software engineering—programming environments. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 259–276. MIT Press, Cambridge, MA, 1998.

[52] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10(1):110–147, January 2001.

[53] G. Rothermel, L. Li, C. Dupuis, and M. Burnett. What You See Is What You Test: A methodology for testing form-based visual programs. In *Proceedings of the* $20^{th}$ *International Conference on Software Engineering*, pages 198–207, June 1998.

[54] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, and G. Rothermel. An empirical evaluation of a methodology for testing spreadsheets. In *Proceedings of the* $22^{nd}$ *International Conference on Software Engineering*, pages 230–239, June 2000.

[55] 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 the ACM Symposium on Software Visualization*, pages 123–132, San Diego, CA, June 11–13, 2003.

[56] J. Sajanieme. Modeling spreadsheet audit: A rigorous approach to automatic visualization. *Journal on Visual Languages and Computing*, 11(1):49–82, 2000.

[57] T. Smedley, P. Cox, and S. Byme. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *ACM Workshop on Advanced Visual Interfaces*, pages 148–155, May 1996.

[58] Telcordia Technologies. xSlice: A tool for program debugging, July 1998. `http://xsuds.argreen-house.com/html-man/coverpage.html`.

[59] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A syntax directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[60] F. Tip. A survey of program slicing techniques. *Journal on Programming Languages*, 3(3):121–189, 1995.

[61] G. Viehstaedt and A. Ambler. Visual representation and manipulation of matrices. *Journal of Visual Languages and Computing*, 3(3):273–298, September 1992.

[62] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[63] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 305–312, Fort Lauderdale, FL, April 5–10, 2003.