MARGARET BURNETT, GREGG ROTHERMEL, AND
CURTIS COOK

AN INTEGRATED SOFTWARE ENGINEERING APPROACH FOR END-USER PROGRAMMERS

ABSTRACT
End-user programming has become the most common form of programming in use today. Despite this growth, there has been little investigation into the correctness of the programs end users create. We have been investigating ways to address this problem, via a holistic approach we call *end-user software engineering*. The concept is to bring support for aspects of software development that happen beyond the "coding" stage—such as testing and debugging—together into the support that already exists for incremental, interactive programming by end users. In this paper, we present our progress on three aspects of end-user software engineering: systematic "white box" testing assisted by automatic test generation, assertions in a form of postconditions that also serve as preconditions, and fault localization. We also present our strategy for motivating end-user programmers to make use of the end-user software engineering devices.

1 INTRODUCTION

There has been considerable work in empowering end users to be able to write their own programs, and as a result, end users are indeed doing so. In fact, the number of end-user programmers—creating programs using such devices as special-purpose scripting languages, multimedia and web authoring languages, and spreadsheets—is expected to reach 55 million by 2005 in the U.S. alone [Boehm et al. 2000]. Unfortunately, evidence from the spreadsheet paradigm, the most widely used of the end-user programming languages, abounds that end-user programmers are extremely prone to introducing faults[1] into their programs [Panko 1998]. This problem is serious, because although some end users' programs are simply explorations and scratch pad calculations, others can be quite important to their personal or business livelihood, such as programs for calculating income taxes, e-commerce web pages, and financial forecasting.

We would like to reduce the fault rate in the end-user programs that are important to the user. Although classical software engineering methodologies are

[1] Following standard terminology, in this paper we use the term "failure" to mean an incorrect output value given the inputs, and the term "fault" to mean the incorrect part of the program (formula) that caused the failure.

Preprint for: *End User Development,* H. Lieberman, F. Paterno, V. Wulf (ed.), pp. 87—113.

not a panacea, there are several that are known to help reduce programming faults, and it would be useful to incorporate some of those successes in end-user programming. Toward this end, we have been working on a vision we call *end-user software engineering*, a holistic approach to end-user software development tasks beyond program entry. Its goal is to bring some of the gains from the software engineering community to end-user programming environments, *without* requiring training or even interest in traditional software engineering techniques.

Our approach to end-user software engineering draws from the traditional software engineering methodologies and programming language literature to devise ways for the system to take on much of the software engineering reasoning, and draws from HCI to find ways to effectively collaborate with the user about the results of this reasoning. End-user software engineering is a highly integrated and incremental concept of software engineering support for end users. Hence, its components are not individual tools, each with a button that can be separately invoked, but rather a blend of knowledge sources that come together seamlessly. At present there are three components that have been blended in, but the overall vision is not restricted to any particular set of components. A continually evolving prototype of the end-user software engineering concept exists for Forms/3 [Burnett et al. 2001a], a research language that follows the spreadsheet paradigm. The components we have so far blended into Forms/3 involve (1) systematic testing, (2) assertions, and (3) fault localization. Portions of the concept have also been extended to the dataflow paradigm [Karam and Smedley 2001] and to the screen transition paradigm [Brown et al. 2003].

2 RELATED WORK

The software engineering research community's work regarding testing, assertions, and fault localization lies at the heart of our approach, but the prerequisite knowledge required to use traditional software engineering approaches is not a good match for end users' skills. Further, the algorithms behind traditional approaches do not usually allow the immediate semantic feedback end-user environments generally provide. These two factors are the reasons it would not be viable to simply import traditional software engineering techniques into an end-user programming environment.

Programming is a collection of problem-solving activities, and our goal is to help end users in these activities. Hence, we draw heavily on HCI research about human problem-solving needs. The HCI research with the greatest influence on our work so far has been Blackwell's theory of Attention Investment [Blackwell 2002], Green et al.'s work on Cognitive Dimensions [Green and Petre1996], Pane et al.'s empirical work [Pane et al. 2002], and psychologists' findings about how curiosity relates to problem-solving behavior [Lowenstein 1994]. Other strong influences have come from the extensive work on end-user and visual programming languages (e.g., [Heger et al. 1998, Igarashi et al. 1998, Lieberman 2001, McDaniel and Myers 1999, Nardi 1993, Repenning and Ioannidou 1997]).

Since end-user software engineering is tightly intertwined with the programming process, it feeds into and draws from the design of languages themselves. For example, immediate visual feedback about semantics is expected in end-user programming, and any software engineering methods that we introduce must be able to reason about the source code efficiently enough to maintain this characteristic. Classic programming language research has contributed to our design of algorithms that provide feedback efficiently, so as to satisfy this constraint. For example, in the language in which we prototype our work, we rely extensively on lazy evaluation [Henderson and Morris 1976] and lazy memoization [Hughes 1985] to keep the incremental costs of each user action small.

In the arena of research into end-user software development, most work to date has concentrated primarily on the "programming" phase (i.e., assisting the user in the process of creating a program). However, work has begun to emerge with the goal of assisting end users in assessing or improving the correctness of their programs. For example, there are several interactive visual approaches related to program comprehension for debugging purposes for both professional and end-user programmers that have made important contributions in this direction. ZStep [Lieberman and Fry 1998] provides visualizations of the correspondences between static program code and dynamic program execution. ZStep also offers a navigable visualization execution history that is similar to features found in some visual programming languages such as KidSim/Cocoa/Stagecast [Heger et al. 1998] and Forms/3 [Burnett et al. 2001a]. S2 [Sajanieme 2000] 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 earlier [Davis 1996].

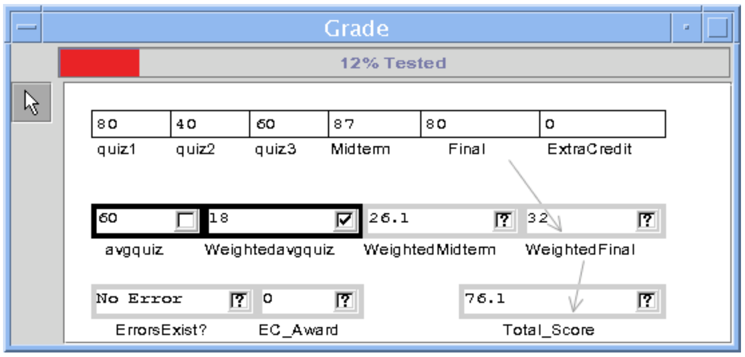
More recently, work aimed particularly at aiding end-user programmers in some software engineering tasks is beginning to emerge. Myers and Ko recently proposed research in assisting users in the debugging of code for event-based languages [Myers and Ko 2003]. Woodstein, an early prototype of an e-commerce debugging tool, is an emerging approach aimed at end-user debugging of actions that go awry in e-commerce transactions [Wagner and Lieberman 2003]. Outlier finding [Miller and Myers 2001] is a method of using statistical analysis and interactive techniques to direct end-user programmers' attention to potentially problematic areas during automation tasks. In this work, outlier finding is combined with visual cues to indicate abnormal situations while performing search and replace or simultaneous editing tasks. Raz et al. also use outlier finding, but in the setting of a particular type of end-user programs, namely web programs that incorporate on-line data feeds [Raz et al. 2002]. Tools have also been devised to aid spreadsheet users in dataflow visualization and editing tasks (e.g., [Igarashi et al. 1998]). reMIND+ [Carr 2003] is 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. There is also some investigation into requirements specification by end users; early work in this direction is described in [Nardi 1993], and there is more recent work on deriving models from informal scenarios [Paterno and Mancini 1999].

3 WYSIWYT TESTING

One of the components of end-user software engineering is the *What You See Is What You Test (WYSIWYT)* methodology for testing [Rothermel et al. 1998, Rothermel et al. 2001, Burnett et al. 2002]. WYSIWYT incrementally tracks test adequacy as users incrementally edit, test and debug their formulas as their programs evolve. All testing-related information is kept up-to-date at all times, and is communicated via the spreadsheet itself through integrated visualization devices.

For example, Figure 1 presents an example of WYSIWYT in Forms/3[2]. In WYSIWYT, untested spreadsheet output (i.e. non-constant) cells are given a red border (light gray in this paper), indicating that the cell is untested. For example, the EC_Award cell has never been tested; hence, its border is red (light gray). The borders of such cells remain red until they become more "tested".

Whenever a user notices that her real or experimental input values are resulting in correct outputs, she can place a checkmark (√) in the decision box at the corner of the cells she observes to be correct: this constitutes a successful *test*. These 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, the user has checked off the Weightedavgquiz cell in Figure 1, 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 has also turned blue (black).



80 quiz1
40 quiz2
60 quiz3
87 Midterm
80 Final
0 ExtraCredit
60 avgquiz
18 Weightedavgquiz
26.1 WeightedMidterm
32 WeightedFinal
No Error ErrorsExist?
0 EC_Award
76.1 Total_Score

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

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.

[2] In the figures, cells are not locked into a grid. One difference between Forms/3 and commercial spreadsheet systems is that Forms/3 allows "free-floating" cells rather than requiring all cells to reside in grids. (This difference also exists in some other research spreadsheet systems, including Forms/2 [Ambler and Burnett 1990] and NoPumpG [Lewis 1990].)

Testing a program "perfectly" (i.e., well enough to detect all possible faults) is not possible without an infinite number of test cases for most programs, so a way is needed to decide when to stop testing. Serving this need, a *test adequacy criterion* is a quantitative measure used to define when a program has been tested "enough". In the spreadsheet paradigm, we say that a cell is fully tested if all its interrelationships (as defined below) have been covered by tests, and those cells are the ones whose borders are painted blue (black). If only some have been covered, the cell is partially tested, and these partially tested cells would have borders in varying shades of purple (shades of gray). Also, to provide some guidance about how to make additional testing progress, if checking off a particular value will increase coverage, that cell's decision box contains a "?".

3.1 Behind the Scenes: The Basis for WYSIWYT's Reasoning

In order to precisely define a test adequacy criterion for use in spreadsheets, we defined an abstract model for spreadsheets, called a *cell relation graph* (CRG). We use CRGs to model those spreadsheets and to define and support testing. A CRG is a pair (V, E), where V is a set of *formula graphs*, and E is a set of directed edges called *cell dependence edges* connecting pairs of elements in V. Each formula graph in V represents the formula for a cell, and each edge in E models the data dependencies between a pair of cells. In the basic form of WYSIWYT, there is one formula graph for each cell in the spreadsheet[3]. Each formula graph models flow of control within a cell's formula, and is comparable to a control flow graph representing a procedure in an imperative program [Aho et al. 1986, Rapps and Weyuker 1985]. Thus, a formula graph is a set of nodes and edges. The nodes in a formula graph consist of an *entry node* modeling initiation of the associated formula's execution, an *exit node* modeling termination of that formula's execution, and one or more *predicate nodes* or *computation nodes*, modeling execution of `if`-expressions' predicate tests and all other computational expressions, respectively. The edges in a formula graph model control flow between pairs of formula graph nodes. The two out-edges from each predicate node are labeled with the values (one true, one false) to which the conditional expression in the associated predicate must evaluate for that particular edge to be taken.

Using the CRG model, we defined a test adequacy criterion for spreadsheets, which we refer to as the *du-adequacy criterion*. We summarize it briefly here; a full formal treatment has been provided elsewhere [Rothermel et al. 2001]. The du-adequacy criterion is a type of dataflow adequacy criterion [Duesterwald et al. 1992, Frankl and Weyuker 1988, Laski and Korel 1993, Rapps and Weyuker 1985]. Such criteria relate test adequacy to interrelationships between definitions and uses of variables in source code. In spreadsheets, cells play the role of variables; a *definition* of cell C is a node in the formula graph for C representing an expression that defines C, and a *use* of cell C is either a *computational use* (a non-predicate node that refers

[3] WYSIWYT has also been extended to reason about multiple cells sharing a single formula [Burnett et al. 2002, Burnett et al. 2001b], but we will not discuss these extensions in this paper.

to *C*) or a *predicate use* (an out-edge from a predicate node that refers to *C*). The interrelationships between these definitions and uses are termed *definition-use associations*, abbreviated *du-associations*. Under the du-adequacy criterion, cell *C* is said to have been *adequately tested* (*covered*) when all of the du-associations whose uses occur in *C* have been *exercised* by at least one test: that is, where inputs have been found that cause the expressions associated with both the definitions and uses to be executed, and where this execution produces a value in some cell that is pronounced "correct" by a user validation. (The closest analogue to this criterion in the literature on testing imperative programs is the *output-influencing-All-du* dataflow adequacy criterion [Duesterwald et al. 1992], a variant of the *all-uses* criterion [Rapps and Weyuker 1985].) In this model, a *test* is a user decision as to whether a particular cell contains the correct value, given the input cells' values upon which it depends.

To facilitate understanding the structure of a spreadsheet, Forms/3 allows the user to pop up dataflow arrows between cells—in Figure 1 the user has triggered the dataflow arrows for the WeightedFinal cell—and even between subexpressions within cell formulas (not shown). The system refers to these as interrelationships in its pop-up tool tips, but the tie to the CRG model is that dataflow arrows between two cells depict the presence of du-associations between those cells. Further, if formulas have been opened on the display (not shown in this figure), the dataflow arrows operate at the granularity of relationships among subexpressions, thus making the du-associations explicit. Dataflow arrows use the same testedness colorization as the cell borders do, so that the users can see exactly which relationships (du-associations) have been tested and which have not. Dataflow arrows for any cell can be displayed and hidden at will by the user.

3.2 "Help Me Test"

Empirical work has shown that the WYSIWYT methodology is helpful to end users[4] [Krishna et al. 2001] but, as presented so far, the WYSIWYT methodology relies on the intuitions of spreadsheet users to devise test cases for their spreadsheets. Even with additional visual devices such as colored arrows between formula subexpressions to indicate the relationships remaining to be covered, after doing a certain amount of testing, users sometimes find it difficult to think of suitable test values that will cover the as-yet-untested relationships. At this point, they can invoke the *Help Me Test (HMT)* feature [Fisher et al. 2002a], to automate the task of input selection.

To see how the approach works, suppose a user creating the spreadsheet in Figure 1 desires help conjuring up test cases that can increase the testedness of that spreadsheet. With HMT, the user selects any combination of cells or dataflow

[4] The particular study referenced involved business students experienced with spreadsheets. Their task was to add new features to a given spreadsheet. We have conducted more than a dozen empirical studies related to the end-user software engineering devices described in this paper. The participants were usually business students, and the tasks were usually testing, debugging, or maintenance. Details can be found at http://www.engr.oregonstate.edu/~burnett/ITR2000/empirical.html.

arrows on the visible display (or, selecting none, signals interest in the entire spreadsheet), and then pushes the "Help Me Test" button on the environment's toolbar.

At this point, the underlying system responds, attempting to generate a test case that will help increase coverage in the user's area of interest. To do this, the system first calculates the set of du-associations that have not yet been tested, that have endpoints within the user's specified area of interest. These du-associations are potential targets for test generation. Next, the system calculates the set of input cells (cells whose formulas are simply constant values, thus serving as the sources of values) that can potentially cause one or more of the target du-associations to be exercised, that is, cause the defining and using subexpressions associated with those du-associations to both be evaluated. Given these two sets of entities (relevant du-associations and relevant input cells) the system can attempt to generate a test case, by manipulating input cells until one or more relevant du-associations can be covered. To perform the manipulations that find the actual test values, our system uses an algorithm adapted from Ferguson and Korel's "Chaining Technique" [Ferguson and Korel 1996].

At any point during these manipulations, if the target du-association is exercised, the system leaves the newly generated values in the cells, and the user can now validate cell values in their area of interest. Alternatively, if they do not like these values (perhaps because the correctness of the particular values is difficult to judge), they can run HMT again to generate different ones. If the process exhausts all possibilities, or reaches an internal time limit, the system informs the user that it has not succeeded. (There is also a "Stop" button available, if the user decides HMT is taking too long.)

Our empirical studies of HMT show that it succeeds in generating test values a high percentage of the time; in one study its success rate always exceeded 94%, and in most cases exceeded 99%. Surprisingly, providing estimates of the ranges in which input values are expected to fall does not improve HMT's success rate. Also, comparisons of HMT to an alternative technique that randomly generates values shows that in all cases HMT was significantly more effective. Response times for HMT were also typically reasonable: in 81% of the trials considered, HMT succeeded in less than 4 seconds, and in 88% of the trials it succeeded in less than 10 seconds. Here, providing range values for input cells can help: with such values available, 91% of responses occurred within 4 seconds, and 96% within 10 seconds.

HMT's efforts to find suitable test values are somewhat transparent to the user—that is, they can see the values it is considering spinning by. The transparency of its behavior turns out to contribute to the understandability of both HMT and assertions. We will return to the tie between HMT and assertions after the next section, which introduces assertions.

4 ASSERTIONS

When creating a spreadsheet, the user has a mental model of how it should operate. One approximation of this model is the formulas they enter. These formulas, however, are only one representation of the user's model of the problem and its solution: they contain information on how to generate the desired result, but do not provide ways for the user to cross-check the computations or to communicate other properties. Traditionally, assertions (preconditions, postconditions, and invariants) have fulfilled this need for professional programmers: they provide a method for making explicit the properties the programmers expect of their program logic, to reason about the integrity of their logic, and to catch exceptions. We have devised an approach to assertions [Burnett et al. 2003, Wilson et al. 2003] that attempts to provide these same advantages to end-user programmers.

Our assertions are composed of Boolean expressions, and reason about program variables' values (spreadsheet cell values, in the spreadsheet paradigm). Assertions are "owned" by a spreadsheet cell. Cell X's assertion is the postcondition of X's formula. X's postconditions are also preconditions to the formulas of all other cells that reference X in their formulas, either directly or transitively through a network of references.

To illustrate the amount of power we have chosen to support with our assertions, we present them first via an abstract syntax. An *assertion* on cell N is of the form:

(N, {*and-assertions*}), where:
each *and-assertion* is a set of *or-assertions*,
each *or-assertion* is a set of (*unary-relation*, *value-expression*) and (*binary-relation*, *value-expression-pair*) tuples,
each *unary-relation* ∈ {=, <, <=, >, >=},
each *binary-relation* ∈ {to-closed, to-open, to-openleft, to-openright},
each *value-expression* is a valid formula expression in the spreadsheet language,
each *value-expression-pair* is two *value-expressions*.

For example, an assertion denoted using this syntax as (N, {{(to-closed, 10, 20), (= 3)}, {= X2}}) means that N must either be between 10 and 20 or equal to 3; and must also equal the value of cell X2.

This abstract syntax is powerful enough to support a large subset of traditional assertions that reason about values of program variables. This abstract syntax follows CNF (Conjunctive Normal Form): each cell's collection of and-assertions, which in turn is composed of or-assertions, is intended to evaluate to true, and hence a spreadsheet's assertion is simply an "and" composition of all cells' assertions.

The abstract syntax just presented is not likely to be useful to end users. Thus, we have developed two concrete syntaxes corresponding to it: one primarily graphical and one textual. The user can work in either or both as desired.

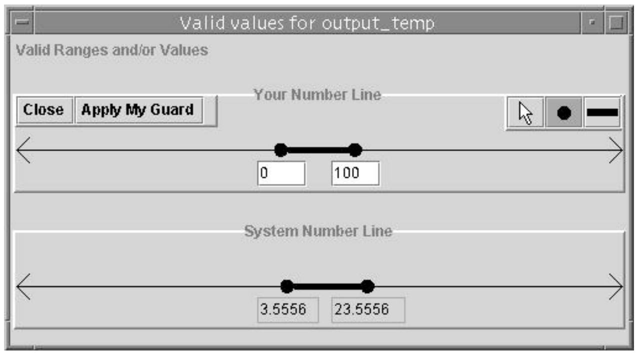


Figure 2. Two (conflicting) assertions "and"ed on the same cell.

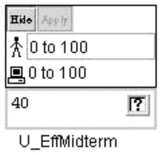


Figure 3. Two assertions in the textual syntax.

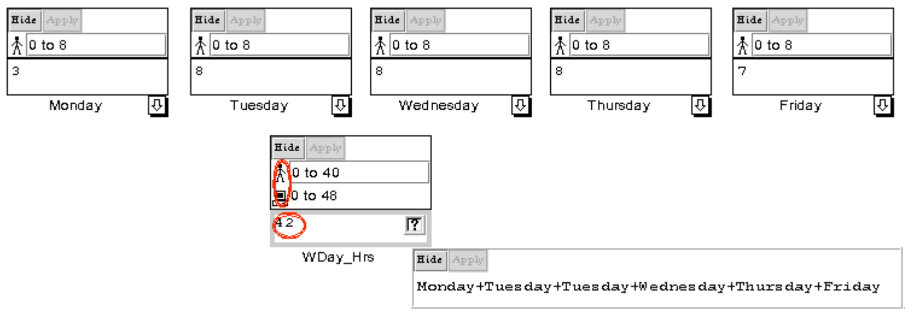
The graphical concrete syntax, depicted in Figure 2, supports all of the abstract syntax (but in the current prototype implementation, value-expressions have been implemented for only constants). The example is a representation of (output_temp, {{(to-closed, 0, 100)}, {(to-closed, 3.5556, 23.5556)}}). A thick dot is a data point in an ordinal domain; it implements "=". The thick horizontal lines are ranges in the domain, implementing "to-closed" when connected to dots. A range with no lower (upper) bound implements "<=" (">="). It is also possible to halve a dot, which changes from closed ranges to open ranges, "<=" to "<", and so on. Disconnected points and ranges represent the or-assertions. Multiple assertions vertically in the same window represent the and-assertions. (We will expand upon how assertions get in and what it means to be "conflicting" shortly.)

The textual concrete syntax, depicted in Figure 3, is more compact, and supports the same operators as the graphical syntax. Or-assertions are represented with comma separators on the same line (not shown), while and-assertions are represented as assertions stacked up on the same cell, as in Figure 3. There is also an "except" modifier that supports the "open" versions of "to" (e.g., "0 to 10 except 10").

Our system does not use the term "assertion" in communicating with users. Instead, assertions are termed *guards*, so named because they guard the correctness of the cells. The user opens a "guard tab" above a cell to display the assertion using the textual syntax, or double-clicks the tab to open the graphical window. Although both syntaxes represent assertions as points and ranges, note that points and ranges, with the composition mechanisms just described, are enough to express the entire abstract syntax.

Note that although "and" and "or" are represented, they are not explicit operators in the syntaxes. This is a deliberate choice, and is due to Pane et al.'s research, which showed that end users are not successful at using "and" and "or" explicitly as logical operators [Pane et al. 2002].

Assertions protect cells from "bad" values, i.e., from values that disagree with the assertion(s). Whenever a user enters an assertion (a *user-entered assertion*) it is propagated as far as possible through formulas, creating *system-generated assertions* on downstream cells. The user can use tabs (not shown) to pop up the assertions, as has been done on all cells in Figure 4. The stick figure icons on cells Monday, Tuesday, ... identify the user-entered assertions. The computer icon on cell WDay_Hrs identifies a system-generated assertion, which the system generated by propagating the assertions from Monday, Tuesday, …, through WDay_Hrs's formula. A cell with both a system-generated and user-entered assertion is in a conflict state (has an *assertion conflict*) if the two assertions do not match exactly. The system communicates an assertion conflict by circling the conflicting assertions in red. In Figure 4 the conflict on WDay_Hrs is due to a fault in the formula (there is an extra Tuesday). Since the cell's value in WDay_Hrs is inconsistent with the assertions on that cell (termed a *value violation*), the value is also circled. When conflicts or violations occur, there may be either a fault in the program (spreadsheet formulas) or an error in the assertions. In [Burnett et al. 2003] we report the results of an empirical study that measured whether assertions contributed to end user programmers' debugging effectiveness; the results were that the participants using assertions were significantly more effective at debugging than were participants without access to assertions.



Monday+Tuesday+Tuesday+Wednesday+Thursday+Friday

Figure 4. In the Forms/3 environment, cell formulas can be displayed via the tab at the lower right hand side of the cell, as has been done in WDay_Hrs. The assertions on each cell have been popped up at the top of the cells.

4.1 Assertions: Detailed Example

We close this section by presenting a detailed example of our prototype assertion mechanism. Figure 5 (top left) shows a portion of a Forms/3 spreadsheet that converts temperatures in degrees Fahrenheit to degrees Celsius. The input_temp cell has a constant value of 200 in its formula and is displaying the same value. There is a user assertion on this cell that limits the value of the cell to between 32 and 212. The formulas of the a, b, and output_temp cells each perform one step in the conversion, first subtracting 32 from the original value, then multiplying by five and finally dividing by nine. The a and b cells have assertions generated by the system (as indicated by the computer icon) which reflect the propagation of the user assertion on the input_temp cell through their formulas. The spreadsheet's creator has told the system that the output_temp cell should range from 0 to 100, and the system has agreed with this range. This agreement was determined by propagating the user assertion on the input_temp cell through the formulas and comparing it with the user assertion on the output_temp cell.

Suppose a user has decided to change the direction of the conversion and make the spreadsheet convert from degrees Celsius to degrees Fahrenheit. A summary follows of the behavior shown by an end user in this situation in a think-aloud study we conducted early in our design of the approach [Wallace et al. 2002]. The quotes are from a recording of that user's commentary.

First, the user changed the assertion on input_temp to range from 0 to 100. This caused several red violation ovals to appear, as in Figure 5 (top right), because the values in input_temp, a, b, and output_temp were now out of range and the assertion on output_temp was now in conflict with the previously specified assertion for that cell. The user decided "that's OK for now," and changed the value in input_temp from 200 to 75 ("something between zero and 100"), and then set the formula in cell a to "input_temp * 9/5" and the formula in cell b to "a + 32".

At this point, the assertion on cell b had a range from 32 to 212. Because the user combined two computation steps in cell a's formula (multiplication and division), the correct value appeared in cell b, but not in output_temp (which still had the formula "b / 9"). The user now chose to deal with the assertion conflict on output_temp, and clicked on the guard icon to view the details in the graphical syntax (refer back to Figure 2).

Seeing that the Forms/3 assertion specified 3.5556 to 23.556, the user stated "There's got to be something wrong with the formula" and edited output_temp's formula, making it a reference to cell b. This resulted in the value of output_temp being correct, although a conflict still existed because the previous user assertion remained at 0 to 100. Turning to the graphical syntax window, upon seeing that Forms/3's assertion was the expected 32 to 212, the user changed the user assertion to agree, which removed the final conflict. Finally, the user tested by trying 93.3333, the original output value, to see if it resulted in approximately 200, the original input value. The results were as desired, and the user checked off the cell to notify the system of the decision that the value was correct, as in Figure 5 (bottom).

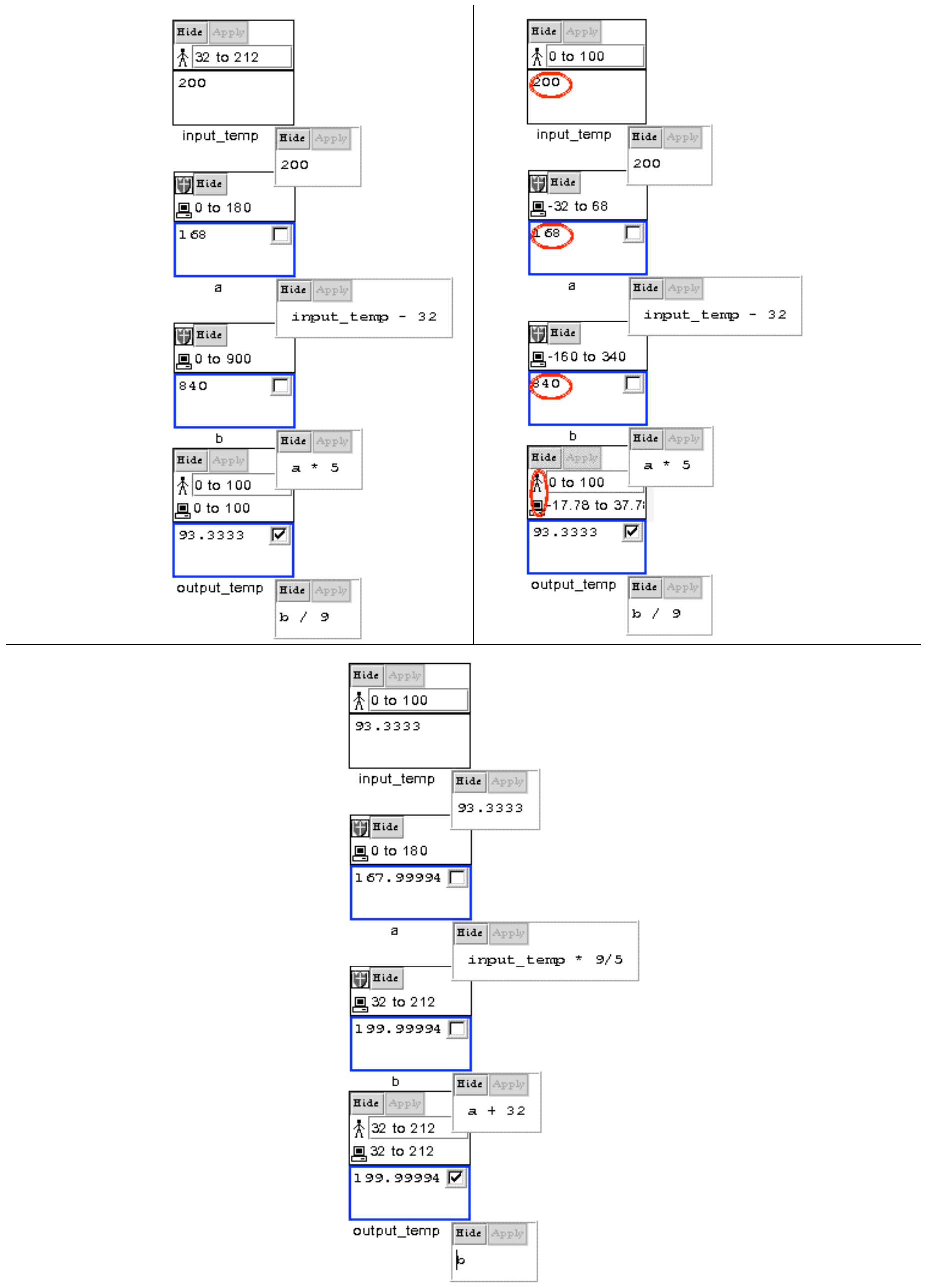


Figure 5. Example at three points in the modification task.

5 IF WE BUILD IT, WILL THEY COME?

Of course, the benefits of assertions can be realized only if users can be enticed into entering their own assertions and acting on them. In the studies on assertions alluded to above, we introduced assertions to our experiment participants via short tutorial sessions. But, without such introductions, will users choose to enter assertions?

Blackwell's model of attention investment [Blackwell 2002] is one model of user problem-solving behavior that suggests users will not want to enter assertions. 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 exploring an unknown feature such as assertions has cost, benefit, and risk. The cost is figuring out what assertions do and how to succeed at them. The benefit of finding faults may not be clear until long *after* the user proceeds in this direction. The risk is that going down this path will be a waste of time or, worse, will leave the spreadsheet in a state from which it is hard to recover. What the model of attention investment implies is that it is necessary not only for our strategy to make the users curious about assertions, but also to make the benefits of using assertions clear from the outset.

In this section, we describe a strategy for doing so. We describe the strategy in the context of assertions, but we intend to eventually generalize it as a way to motivate users toward *any* appropriate software engineering devices in the environment. We term our strategy the *Surprise-Explain-Reward* strategy [Wilson et al. 2003]. The strategy draws on the model of attention investment and on findings about the psychology of curiosity. As the name suggests, the strategy consists of three components: a collection of surprises, a collection of rewards, and an explanation component pointing out the links from the surprises to the rewards.

Research about curiosity (surveyed in [Lowenstein 1994]), points out that if an information gap is illustrated to the user, the user's curiosity about the subject of the illustrated gap may increase, potentially causing them to search for an explanation. Without challenging their assumptions and arousing their curiosity, as the information-gap perspective explains and much empirical programming literature bears out (e.g., [Panko 1998, Wilcox et al. 1997, Krishna et al. 2001]), users are likely to assume that their programs are more correct than is warranted. This is the motivation for the first component of our Surprise-Explain-Reward strategy: to arouse users' curiosity, through surprise, enough that they search for explanations. Thus, the first component of our strategy might be characterized as following a "principle of *most* astonishment."

The strategy is used in two situations: first to entice users to use the features, and later to help them benefit from the features. In the first situation, the strategy attempts to surprise the user into entering assertions. If the user becomes curious about the assertions, she can find out more via an explanation system. If the strategy causes the user to act by entering an assertion, the second situation becomes possible. This time, the surprise comes when an assertion identifies a potentially faulty cell formula. (Actually, the assertion identifies a failure rather than a fault, but since even intermediate cells in the dataflow chain can be monitored by assertions, the probability of the fault and failure being in the same cell is greater than would be the case if only final outputs were monitored.) The users can again look to explanations to explain the surprise and suggest possible actions. If the user successfully fixes the fault called to their attention by the assertion, they see that the program's behavior is more correct, a clear reward for using assertions. In the remainder of this section we expand upon the approach we have just summarized.

5.1 Surprises

The first step of our strategy is to generate a meaningful surprise for the user. That is, the system needs to violate the user's assumptions about their program. We have devised a pseudo-assertion for this purpose, termed an *HMT assertion* because it is produced by the "Help Me Test" (HMT) device described earlier. An HMT assertion is a guess at a possible assertion for a particular cell.

The guesses are actually reflections of HMT's behavior. That is, they report the range of HMT's attempts to find suitable test cases. For example, in Figure 6 (which is part of the weekly payroll program of Figure 4), the HMT assertion for cell Monday is "-1 to 0." This indicates that HMT has considered values for Monday between -1 and 0 before it settled upon its current value of -1. If HMT is invoked again it might consider a different selection of values for Monday such as 1 and 2, which would widen the HMT assertion to "-1 to 2." (Note that this tie between HMT's test case generation behavior and the assertions it guesses creates a reward opportunity for manipulating the assertions.) The primary job of the HMT assertions is to surprise the user, and thereby to generate user interest in "real" assertions (i.e., user-entered and system-generated assertions). Thus, HMT assertions are—by design—usually bad guesses. The worse the guess, the bigger the surprise.

HMT assertions exist to surprise and thereby to create curiosity. For example, in Figure 6, the user may expect values for Monday to range from 0 to 8, and rightly so, because employees cannot be credited with fewer than 0 or more than 8 hours per day. Since HMT was not aware of this, it attempted inputs less than zero. Thus, the HMT assertion for Monday probably violates the user's assumptions about the correct values for Monday. This is precisely what triggers curiosity according to the information-gap perspective.

Once an HMT assertion has been generated, it behaves as any assertion does. Not only does it propagate, but if a value arrives that violates it, the value is circled in red. This happens even as HMT is working to generate values. Thus, red circles sometimes appear as HMT is doing its transparent search for suitable test cases. These red circles are another use of surprise.

It is important to note that, although our strategy rests on surprise, it does not attempt to rearrange the user's work priorities by requiring users to do anything about the surprises. No dialog boxes are presented and there are no modes. HMT assertions are a passive feedback system; they try to win user attention but do not require it. If users choose to follow up, they can mouse over the assertions to receive an explanation, which explicitly mentions the rewards for pursuing assertions. In a behavior study we performed [Wilson et al. 2003], users did not always attend to

HMT assertions for the first several minutes of their task; thus it appears that the amount of visual activity is reasonable for requesting but not demanding attention. However, almost all of them did eventually turn their attention to assertions, and when they did, they used assertions effectively.

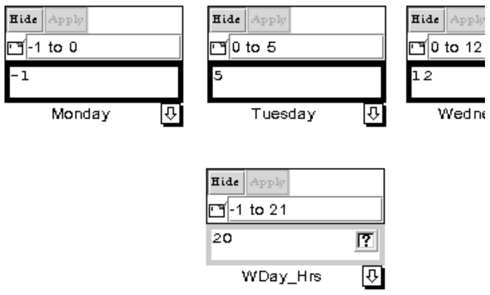


Figure 6. HMT has guessed assertions for the input cells (top row). (Since HMT changed the values of the input cells, they are highlighted with a thicker border.) The guesses then propagated through WDay_Hrs's formula to create an HMT assertion for that cell as well.

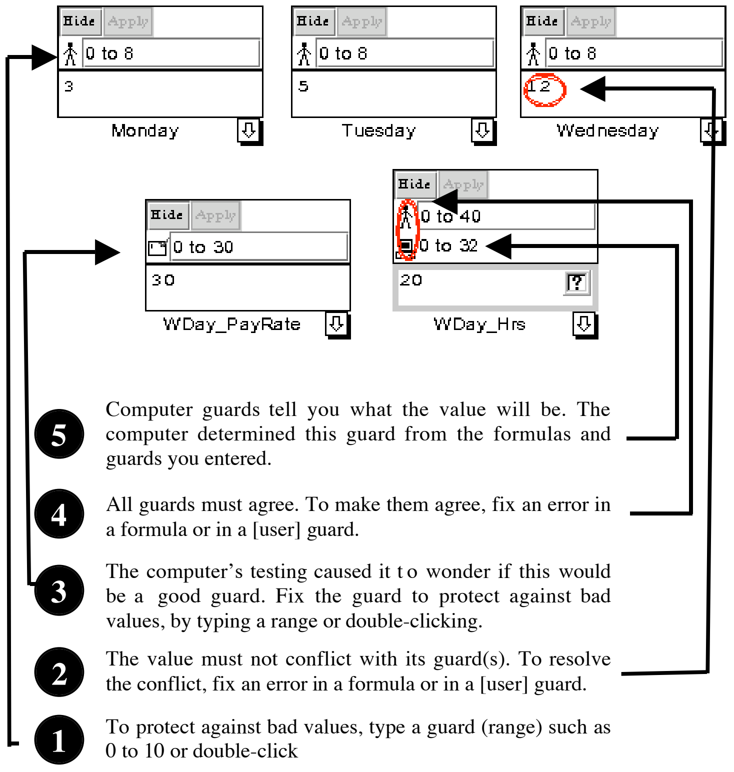
5.2 Explanations

As the above paragraph implies, our strategy's explanation component rests upon self-directed exploration, following in the direction advocated by several researchers who have empirically studied this direction and have found it to result in superior learning and performance of computer tasks (e.g., [Carroll and Mack 1984, Wulf and Golombek 2001]). To support self-directed exploration, in our strategy a feature that surprises a user must inform the user. As the second component of our Surprise-Explain-Reward strategy, we devised an on-demand explanation system structured around each object that might arouse curiosity. Users can begin exploring the object by viewing its explanation, on demand, in a low-cost way via tool tips.

For example, when a user mouses over an HMT assertion they receive explanation 3 in Figure 7. The explanation describes the semantics: the computer was responsible for creating this assertion, and the assertion was a product of the computer's testing. The end of the explanation suggests a possible action for the user to try (fixing the assertion) and specifies a potential reward (protecting against bad values).

Note that the computer "wonders" about this assertion. This makes explicit that the HMT assertion may not be correct and that the computer would like the user's advice. Previous empirical work has revealed that some users think the computer is always correct (e.g., [Beckwith et al. 2002]). Thus it is important to emphasize the tentative nature of the HMT assertions.

The explanation system spans all the objects in the environment. In general, the three main components of explanations include: the semantics of the object, suggested action(s) if any, and the reward. Including the semantics, action, and reward as part of the explanation are not arbitrary choices. Regarding semantics, although many help systems for end users focus mostly on syntax, a study assessing how end users learn to use spreadsheets found that the successful users focused more on the semantics of the spreadsheet than on syntax [Reimann and Neubert 2000]. Regarding actions, Reimann and Neubert are among those who have examined learning by exploration. They point out that users (novices or experts) often form sub-goals using clues in the environment. The actions in the explanations suggest such sub-goals. Getting the user to take action in order to learn is, in fact, a principle of the minimalist model of instruction [Carroll and Mack 1984, Rosson and Seals 2001, Seals et al. 2002]. Regarding reward, the attention investment model emphasizes the fact that the suggested action will cost the user effort and that, unless the potential rewards are made clear, users may not be able to make an informed decision about whether or not to expend the effort.



5 Computer guards tell you what the value will be. The computer determined this guard from the formulas and guards you entered.

4 All guards must agree. To make them agree, fix an error in a formula or in a [user] guard.

3 The computer's testing caused it to wonder if this would be a good guard. Fix the guard to protect against bad values, by typing a range or double-clicking.

2 The value must not conflict with its guard(s). To resolve the conflict, fix an error in a formula or in a [user] guard.

1 To protect against bad values, type a guard (range) such as 0 to 10 or double-click

Figure 7. Five explanation examples.

5.3 Rewards

When the user edits an HMT assertion to create a user-entered assertion, there are three types of short-term rewards that can follow. There is also a fourth, longer term reward, namely the bridge to "real" assertions and their long-term rewards.

The first reward, which visibly occurs in some situations, is input value validation. This reward can occur immediately when the user edits an HMT assertion. Consider again cell Monday in Figure 6. Suppose the user notices the HMT assertion, reads explanation 3 from Figure 7 and, deciding to take the explanation's advice to fix the assertion, changes it to "0 to 8." (The HMT assertion helps show how to succeed by acting as a template, exemplifying assertion syntax.) Despite the assertion entry, the cell's *value* is still –1, and the system circles the value, since it is now in violation with its assertion. Thus, by taking the advice of the system the user has been immediately rewarded: the system is indeed "protect(ing) against bad values."

The second reward always occurs. Once a user places an assertion on an input cell, the behavior of HMT changes to honor the assertion. Continuing the previous example, once cell Monday has the assertion "0 to 8" and the user runs HMT again, HMT will always choose values satisfying the assertion. Since HMT's "thought process" is displayed as it mulls over values to choose, this behavior change is noticeable to some users. Since HMT's selected values can seem odd from the user's perspective, given their knowledge of the program's purpose, getting HMT to choose sensible input values is rewarding if noticed. In our empirical work, a few of the participants' comments showed that they not only noticed this tie but that the tie was what motivated them to use assertions.

The third reward also pertains to changes in HMT's behavior. HMT becomes an aggressive seeker of test values that will expose faults. As other test generators in the software engineering community have done (e.g., [Korel and Al-Yami 1996]), HMT attempts to violate user-entered or system-generated assertions. This behavior is focused on non-input cells (i.e., cells that have formulas instead of constant values), and potentially creates a value violation. A value violation on a non-constant cell indicates one of three things: an erroneous assertion, a situation in which the program could fail given inappropriate values in upstream input cells not protected by assertions, or the presence of a faulty formula. For example, for cell WDay_Hrs, HMT will attempt to violate the assertion "0 to 40" by looking for values in the inputs contributing to WDay_Hrs that produce a value violation. When HMT's pursuit of faults succeeds, the user is not only rewarded, but also is probably surprised at the presence of the heretofore unnoticed fault, leading to a longer term use of the Surprise-Explain-Reward strategy.

HMT assertions are intended to help users learn and appreciate assertions, but after that goal has been accomplished, some users will not need HMT's guessed assertions. They will have learned how to enter assertions, regardless of whether HMT guesses assertions on the cells they wish to protect. The Surprise-Explain-Reward strategy carries over to a longer term, to help maintain correctness.

Assertions can lead to three kinds of surprises, and these surprises are themselves rewards, since they mean a fault has been semi-automatically identified. First, the value violations are surprises. As already discussed, they identify faults or assertion errors, and are circled in red. Second, assertion conflicts are surprises. As explained earlier, assertion conflicts arise when the system's propagating the user-entered assertions through formulas produce system-generated assertions that disagree with user-entered assertions. Like value violations, they are circled in red (as in WDay_Hrs in Figure 4 and Figure 7) and indicate faults or assertion errors. Third, the system-generated assertions might "look wrong" to the user.

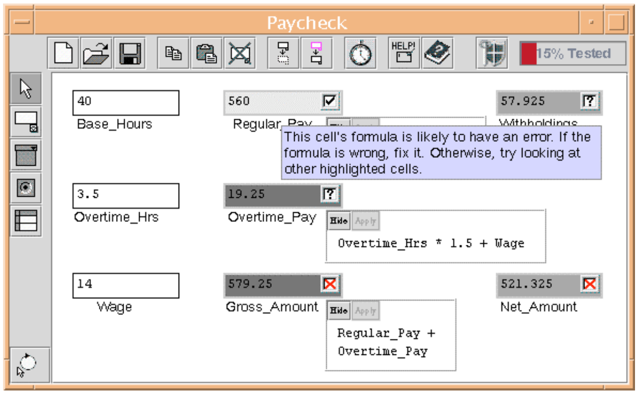
Our behavior study, which is detailed in [Wilson et al. 2003], provided two types of evidence that the rewards were indeed sufficient to convince the participants of the value of assertions. The strongest and most general is the fact that, when participants used assertions once, they used them again. (The mean was 18 assertions per participant.) Additional evidence is that, although participants did not enter assertions until almost 14 minutes (mean) into the first task, by the second task they began entering assertions much earlier[5]. In fact, 9 of the 16 users (56%) entered assertions within the first minute after beginning the second task. From this it seems clear that after users became familiar with assertions during the first task, they had learned to recognize their value by the second task. Further, all introduction to the assertions took place through the Surprise-Explain-Reward strategy itself: our experiment's tutorial never mentioned assertions or gave any insights into what they were or how to use them.

6 FAULT LOCALIZATION

Given the explicit, visualization-based support for WYSIWYT testing, an obvious opportunity is to leverage the users' testing information to help with fault localization once one of their tests reveals a failure. Our end-user software engineering work takes advantage of this opportunity through the use of slicing.

While incrementally developing a spreadsheet, a user can indicate his or her observation of a failure by marking a cell incorrect with an "X" instead of a checkmark. At this point, our fault localization techniques highlight in varying shades of red the cells that might have contributed to the failure, with the goal that the most likely-to-be faulty cells will be colored dark red. More specifically, each cell's interior is colored with one of six discrete fault likelihood colors so as to highlight the cells most likely to contain faults: "None" (no color), "Very Low," "Low," "Medium," "High," and "Very High" (very dark red). For example, in Figure 8, the fault likelihood of the Regular_Pay cell has been estimated as Very Low, the fault likelihoods of Withholdings and Net_Amount have been estimated as Medium, and the fault likelihoods of Overtime_Pay and Gross_Amount have been estimated as High.

[5] There were two different spreadsheets, but the order they were given to the participants was varied. When we refer to the "first" or "second" task, we mean the first or second (respectively) spreadsheet in that particular participant's sequence.



40
Base_Hours

560
Regular_Pay

57.925
Withholdings

This cell's formula is likely to have an error. If the formula is wrong, fix it. Otherwise, try looking at other highlighted cells.

3.5
Overtime_Hrs

19.25
Overtime_Pay

```
Overtime_Hrs * 1.5 + Wage
```

14
Wage

579.25
Gross_Amount

```
Regular_Pay +
Overtime_Pay
```

521.325
Net_Amount

Figure 8. A Paycheck spreadsheet. The user is hovering the mouse over the Overtime_Pay cell to see the explanation of its dark red shading.

6.1 *Three Techniques for Estimating Fault Likelihood*

How should these colors be computed? Computing exact fault likelihood values for a cell, of course, is not possible. Instead, we must combine heuristics with deductions that can be drawn from analyzing the source code (formulas) and/or from the user's tests. We are currently experimenting with three different approaches to assisting end-user programmers with fault localization. All three techniques maintain the following three properties:

(1) Every cell that might have contributed to the computation of an incorrect value will be assigned some positive fault likelihood.

(2) The more incorrect calculations a cell contributes to, the greater the fault likelihood assigned to it.

(3) The more correct calculations a cell contributes to, the lower the fault likelihood assigned to it.

The first approach, like many other fault localization techniques, builds on previous research on slicing and dicing. (Tip provides a survey of that research [Tip 1995].) In general, a program's backward slice is every portion of the program that affects a particular variable at a particular point. (Similarly, a forward slice is every portion of the program that the particular variable at that point affects.) In the spreadsheet paradigm, the concept of a backward (forward) slice simplifies to every cell whose value contributes to (is affected by) the cell in question. For example, in Figure 8 the backward slice of Gross_Amount is Regular_Pay and Overtime_Pay, plus the three cells used in the computation of those two. Since Gross_Amount is exhibiting a failure, its backward slice contains all the cells that could possibly contain the fault. A dice [Chen and Cheung 1997] can reduce the number of cells being considered as having the fault by "subtracting out" of the slice the cells that contributed also to correct values.

In a manner reminiscent of program dicing, our first technique [Reichwein et al. 1999, Ruthruff et al. 2003], which we term the *Blocking Technique*, considers the dataflow relationships involving the X-marked cells that *reach C*. (An X-mark reaches C if there is a dataflow path from C to the X-marked cell that does not have any checkmarks on the cells in the path. If any such checkmarks are present, they are said to *block* the X-mark from C along that path.) In this technique, the more X-marks that reach C, the greater the fault likelihood, supporting property (2) above. For example, in Figure 8, the checkmark in Regular_Pay blocks most of the effects of the X-marks downstream from affecting Regular_Pay's fault likelihood—but it does not completely block out the X-marks' effects because of property (1) above. Unlike previous algorithms, our algorithms are incremental, updating and reflecting fault likelihood after each user action (each formula edit, each placement of a checkmark or X-mark, etc.). The Blocking Technique is also different from previous algorithms in its use of reasoning about which marks are blocked from *C* by marks of the opposite type in *C's* forward slice.

Another way to reason about slices is via counts of successful/failed tests. Jones et al. developed Tarantula [Jones et al. 2002], which follows that type of strategy. Tarantula utilizes information from all passing and failing tests when highlighting possible locations for faults. It colors each statement to indicate the likelihood that it is faulty, determining colors via the ratio of failing to passing tests for tests that execute that statement. Like xSlice, Tarantula reports its results after running the entirety of a test suite and collecting information from all of the tests in the test suite. Our second technique, which we term the *Test Count Technique* [Fisher et al. 2002b, Ruthruff et al. 2003], follows the same general strategy but, as already mentioned above, our techniques incrementally update information about likely locations for faults as soon as the user applies tests to their spreadsheet.

Both the Blocking and Test Count techniques can be a bit too expensive to be practical for the highly responsive, incremental conventions of end-user programming environments. Under some circumstances, their time complexities approach $O(n^4)$ and $O(n^3)$ respectively, where n is the number of cells in the spreadsheet. (A complete discussion of complexities is given in [Ruthruff et al. 2003].) To address this issue, we devised a third technique that approximates the reasoning from both techniques at a lower cost, namely O(n).

6.2 Which of the Techniques is Best?

We are currently conducting a variety of empirical studies to see which of the fault localization techniques just described is the most effective, given the testing end-user programmers actually do. Our empirical work on this question is still in early stages, so we are not ready to identify which is the "best" of the three techniques. But we can provide the following insights:

- Recall that these techniques are meant for *interactive, incremental* programming. Thus, feedback is needed before there is very much information available, such as after the very first X-mark the user places.

- Early experimental work suggests that the Blocking Technique outperforms the other two in ability to visually discriminate the faulty cells from the non-faulty ones at early stages [Ruthruff et al. 2003].
- In interactively judging the correctness of values, end-user programmers, being human, make mistakes. In a study we conducted [Prabhakararao et al. 2003], approximately 5% of the checkmarks end users placed were incorrect.
- If a fault localization mechanism is not robust enough to tolerate mistakes, it can greatly interfere with the user's debugging success. In the study just mentioned, the 5% mistake rate seriously interfered with over half the participants' debugging sessions (using the Blocking Technique).
- In our empirical work, the Test Count Technique has been more robust than the other two techniques in tolerating a reasonably small mistake rate [Ruthruff et al. 2003]. This is probably because the Test Count Technique is historical in nature. That is, it considers the entire history of test values, allowing the correct information to build up, essentially "outweighing" a small mistake rate. In contrast to this, the Blocking Technique emphasizes how the most recent test cases' set of judgments (checks and X-marks) interact.

In an experiment in which users were first familiarized with the presence of a fault localization mechanism, they tended to make use of it, but only after their own debugging sleuthing failed. When they did eventually turn to a fault localization mechanism, it was often quite helpful at leading them to the fault [Prabhakararao et al. 2003].

However, a puzzling problem we have observed in our empirical work is that, if users have not been made familiar with the presence of the fault localization mechanism, when they eventually encounter it they do not seem to trust it enough to make use of its assistance. This is entirely different from their response to assertions, which they seem to embrace wholeheartedly. We are currently working to learn the reasons for this marked difference in user attitude toward the two mechanisms.

7 CONCLUDING REMARKS

Our view is that giving end-user programmers ways to easily create their own programs is important, but is not enough. We believe that, like their counterparts in the world of professional programming, end-user programmers need support for other aspects of the software lifecycle. In this paper, we have presented our approach to end-user software engineering, which integrates, in a fine-grained way, support for testing, assertions, and fault localization into the user's programming environment. As part of this work, we have also been working on how to motivate end users to make use of the software engineering devices, and have gotten encouraging results via our Surprise-Explain-Reward strategy. Supporting software development activities beyond the programming stage—in a way that helps users make productive use of the devices but does not require them to invest in software engineering training—is the essence of our end-user software engineering vision.

8 ACKNOWLEDGMENTS

This work was supported in part by NSF under ITR-0082265 and in part by the EUSES Consortium via NSF's ITR-0325273. We would like to acknowledge the many students and collaborators who have contributed to the end-user software engineering methodologies and empirical studies described in this paper: Miguel Arredondo-Castro, Laura Beckwith, Darren Brown, Joshua Cantrell, Mingming Cao, Nanyu Cao, Ledah Casburn, Frank Cort, Eugene Rogan Creswick, Christopher DuPuis, Mike Durham, Marc Fisher, Orion Granatir, Thomas Green, Dalai Jin, Daniel Keller, Andrew Ko, Vijay Krishna, John LeHoullier, Lixin Li, Martin Main, Omkar Pendse, Amit Phalgune, Shrinu Prabhakararao, James Reichwein, Bing Ren, T. J. Robertson, Karen Rothermel, Joseph Ruthruff, Justin Schonfeld, Prashant Shah, Andrei Sheretov, Jay Summet, Christine Wallace, and Aaron Wilson.

9 REFERENCES

[Aho et al. 1986] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Ambler and Burnett 1990] A. Ambler and M. Burnett, Visual forms of iteration that preserve single assignment, *Journal of Visual Languages and Computing* 1(2), June 1990, 159-181.

[Beckwith et al. 2002] L. Beckwith, M. Burnett, and C. Cook, Reasoning about many-to-many requirement relationships in spreadsheets, *Proc. IEEE Symp. Human-Centric Computing*, Arlington VA, Sept. 2002, 149-157.

[Blackwell 2002] A. Blackwell, First steps in programming: a rationale for attention investment models, *Proc. IEEE Human-Centric Computing Languages and Environments,* Arlington, VA, Sept. 3-6, 2002, 2-10.

[Boehm et al. 2000] B. Boehm, C. Abts, A. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, J. Reifer, and B. Steece,, *Software Cost Estimation with COCOMO II.* Prentice Hall PTR, Upper Saddle River, NJ, 2000.

[Brown et al. 2003] D. Brown, M. Burnett, G. Rothermel, H. Fujita, and F. Negoro, Generalizing WYSIWYT visual testing to screen transition languages, *Proc. IEEE Symp. Human-Centric Computing Languages and Environments*, Auckland, NZ, Oct. 28-31, 203-210.

[Burnett et al. 2001a] 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. *J. Functional Programming 11* (2), March 2001, 155-206.

[Burnett et al. 2001b] M. Burnett, B. Ren, A. Ko, C. Cook, and G. Rothermel, Visually testing recursive programs in spreadsheet languages, *Proc. IEEE Human-Centric Computing Languages and Environments,* Stresa, Italy, Sept. 5-7, 2001, 288-295.

[Burnett et al. 2002] M. Burnett, A. Sheretov, B. Ren, and G. Rothermel, Testing homogeneous spreadsheet grids with the 'What You See Is What You Test' methodology, *IEEE Trans. Software Engineering*, May 2002, 576-594.

[Burnett et al. 2003] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace, End-user software engineering with assertions in the spreadsheet paradigm, *Proc. Int'l. Conf. Software Engineering*, Portland, OR, May 3-10, 2003, 93-103.

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

[Carroll and Mack 1984] J. Carroll and R. Mack, Learning to use a word processor by doing, by thinking, by knowing. In *Human Factors in Computer Systems* (J. C. Thomas and M. L. Schneider, eds.), Ablex, Norwood, NJ, 1984, 13-51.

[Chen and Cheung 1997] T. Chen and Y. Cheung, On program dicing. *Software Maintenance: Research and Practice* 9, 1, Jan.–Feb. 1997, 33–46.

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

[Duesterwald et al. 1992] E. Duesterwald, R. Gupta, and M. L. Soffa, Rigorous data flow testing through output influences, *Proc. Second Irvine Software Symposium*, Mar. 1992, 131-145.

[Ferguson and Korel 1996] R. Ferguson and B. Korel, The chaining approach for software test generation, *ACM Trans. Software Engineering and Methodology* 5(1), Jan. 1996, 63-86.

[Fisher et al. 2002a] M. Fisher, M. Cao, G. Rothermel, C. Cook, and M. Burnett, Automated test generation for spreadsheets, *Proc. Int. Conf. Software Engineering*, Orlando FL, May 2002, 141-151.

[Fisher et al. 2002b] M. Fisher, D. Jin, G. Rothermel, and M. Burnett, Test reuse in the spreadsheet paradigm. *Proc. Int. Symp. Software Reliability Engineering*, Nov. 2002.

[Frankl and Weyuker 1988] P. Frankl and E. Weyuker, An applicable family of data flow criteria, *IEEE Trans. Software Engineering* 14(10), Oct. 1988, 1483-1498.

[Green and Petre1996] T. R. G. Green and M. Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing 7*(2), June 1996, 131-174.

[Heger et al. 1998] N. Heger, A. Cypher, and D. Smith, Cocoa at the Visual Programming Challenge 1997, *J. Visual Langs. and Computing* 9(2), Apr. 1998, 151-168.

[Henderson and Morris 1976] P. Henderson and J. Morris, A lazy evaluator, *Proc. ACM Symp. on Principles of Programming Languages*, Atlanta, GA, Jan. 19-21, 1976, 95-103.

[Hughes 1985] J. Hughes, Lazy memo-functions, LNCS #201, (Jean-Pierre Jouannaud, ed.), *Functional Programming Languages and Computer Architecture*, Nancy, France, September 16-19, 1985, 129-146.

[Igarashi et al. 1998] T. Igarashi, J. Mackinlay, B.-W. Chang, and P. Zellweger, Fluid visualization of spreadsheet structures, *Proc. IEEE Symp. Visual Langs.*, 1998, 118-125.

[Jones et al. 2002] J. Jones, M. Harrold, and J. Stasko, Visualization of test information to assist fault localization, *Proc. International Conf. Software Engineering*, Orlando FL, May 2002, 467-477.

[Karam and Smedley 2001] M. Karam and T. Smedley, A testing methodology for a dataflow based visual programming language, *Proc. IEEE Symp. Human-Centric Computing Languages and Environments,* Stresa, Italy, Sept. 5-7, 2001, 280-287.

[Korel and Al-Yami 1996] B. Korel and A. Al-Yami, Assertion-oriented automated test data generation, *Proc. International Conf. Software Engineering,* Berlin Germany, March 1996, 71-80.

[Krishna et al. 2001] 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, *Proc. Int. Conf. Software Maintenance*, Florence Italy, Nov. 2001, 72-81.

[Laski and Korel 1993] J. Laski and B. Korel, A data flow oriented program testing strategy. *IEEE Trans. Soft. Eng. 9*(3), May 1993, 347-354.

[Lewis 1990] C. Lewis, NoPumpG: Creating interactive graphics with spreadsheet machinery, In *Visual Programming Environments: Paradigms and Systems (*Ephraim P. Glinert, ed.), IEEE Computer Society Press, Los Alamitos, CA, 1990, 526-546.

[Lieberman 2001] H. Lieberman, *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, San Francisco, CA, 2001.

[Lieberman and Fry 1998] H. Lieberman and C. Fry, ZStep 95: a reversible, animated source code stepper, In *Soft. Visualization: Programming as a Multimedia Experience* (J. Stasko, J. Domingue, M. Brown, and B. Price, eds.), MIT Press, Cambridge, MA, 1998, 277-292.

[Lowenstein 1994] G. Lowenstein, The psychology of curiosity. *Psychological Bulletin 116*(1), 1994, 75-98.

[McDaniel and Myers 1999] R. McDaniel and B. Myers, Getting more out of programming-by-demonstration, *Proc. ACM Conference on Human Factors in Computing Systems*, Pittsburgh, PA, May 15-20, 1999, 442-449.

[Miller and Myers 2001] R. Miller and B. Myers, Outlier finding: focusing user attention on possible errors, *Proc. User Interface Software and Technology*, Orlando FL, Nov. 2001, 81-90.

[Myers and Ko 2003] B. Myers and A. Ko, Studying Development and debugging to help create a better programming environment, *Proc. CHI 2003 Workshop on Perspectives in End User Development,* April 2003, 65-68

[Nardi 1993] B. Nardi, *A Small Matter of Programming: Perspectives on End-User Computing*, MIT Press, Cambridge, MA, 1993.

[Pane et al. 2002] J. Pane, B. Myers, and L. Miller, Using HCI techniques to design a more usable programming system, *Proc. IEEE Human-Centric Computing Languages and Environments*, Arlington VA, September 2002, 198-206.

[Panko 1998] R. Panko, What we know about spreadsheet errors. *J. End User Computing*, Spring 1998.

[Paterno and Mancini 1999] F. Paterno and C. Mancini, Developing task models from informal scenarios, *Proc. ACM CHI'99 Late Breaking Results*, Pittsburgh, PA, 228-220, May 1999.

[Prabhakararao et al. 2003] 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, *Proc. IEEE Symp. Human-Centric Computing Languages and Environments*, Auckland, New Zealand, October 28-31, pp. 15-22.

[Rapps and Weyuker 1985] S. Rapps and E. Weyuker, Selected software test data using data flow information. *IEEE Trans. Soft. Eng. 11*(4), Apr. 1985, 367-375.

[Raz et al. 2002] O. Raz, P. Koopman, and M. Shaw, Semantic anomaly detection in online data sources, *Proc. 24th International Conference on Software Engineering*, Orlando, FL, May 19-25, 2002, 302-312.

[Reichwein et al. 1999] J. Reichwein, G. Rothermel, and M. Burnett, Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. *Proc. 2nd Conf. Domain Specific Languages,* Oct. 1999, 25-38.

[Reimann and Neubert 2000] P. Reimann and C. Neubert, The role of self-explanation in learning to use a spreadsheet through examples. *J. Computer Assisted Learning 16*, Dec. 2000, 316-325.

[Repenning and Ioannidou 1997] A. Repenning and A. Ioannidou, Behavior processors: layers between end-users and Java virtual machines, *1997 IEEE Symp. Visual Languages*, Capri, Italy, September 1997, 402-409.

[Rosson and Seals 2001] M. Rosson and C. Seals, Teachers as simulation programmers: minimalist learning and reuse, *Proc. ACM Conf. Human Factors in Computing Systems,* Seattle, WA, Apr. 2001, 237-244.

[Rothermel et al. 1998] G. Rothermel, L. Li, C. DuPuis, and M. Burnett, What you see is what you test: a methodology for testing form-based visual programs, *Proc .International Conf. Software Engineering* Kyoto Japan, Apr. 1998, 198-207.

[Rothermel et al. 2001] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov, A methodology for testing spreadsheets, *ACM Trans. Software Engineering and Methodology 10(*1), Jan. 2001, 110-147.

[Ruthruff et al. 2003] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main, End-user software visualizations for fault localization, *Proc. ACM Symp. Software Visualization*, San Diego, CA, June 11-13, 2003, 123-132.

[Sajanieme 2000] J. Sajanieme, Modeling spreadsheet audit: a rigorous approach to automatic visualization, *J. Visual Langs. and Computing* 11(1), 2000, 49-82.

[Seals et al. 2002] C. Seals, M. Rosson, J. Carroll, T. Lewis, and L. Colson, Fun learning Stagecast Creator: an exercise in minimalism and collaboration, *Proc. IEEE Symp. Human-Centric Computing Languages and Environments*, Arlington VA, Sept. 2002, 177-186.

[Tip 1995] F. Tip, A survey of program slicing techniques. *J. Programming Languages* 3, 3, 1995, 121-189.

[Wagner and Lieberman 2003] E. Wagner and H. Lieberman, An end-user tool for e-commerce debugging. *Proc. Intelligent User Interfaces,* 2003, Miami, Florida, Jan. 12-15, 2003.

[Wallace et al. 2002] C. Wallace, C. Cook, J. Summet, and M. Burnett, Assertions in end-user software engineering: a think-aloud study (Tech Note), *Proc. IEEE Symp. Human-Centric Computing Languages and Environments*, Arlington VA, September 2002, 63-65.

[Wilcox et al. 1997] E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, and C. Cook, Does continuous visual feedback aid debugging in direct-manipulation programming languages? *Proc. ACM Conf. Human Factors in Computing Systems*, Atlanta GA, March 1997, 258-265.

[Wilson et al. 2003] 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, *Proc. ACM Conf. Human Factors in Computing Systems*, Ft. Lauderdale, FL, Apr. 3-10, 2003, 305-312.

[Wulf and Golombek 2001] V. Wulf and B. Golombek, Exploration environments—concept and empirical evaluation, *Proc. GROUP*, 2001, 107-116.

A. APPENDIX A: WYSIWYT SCENARIOS IN EXCEL

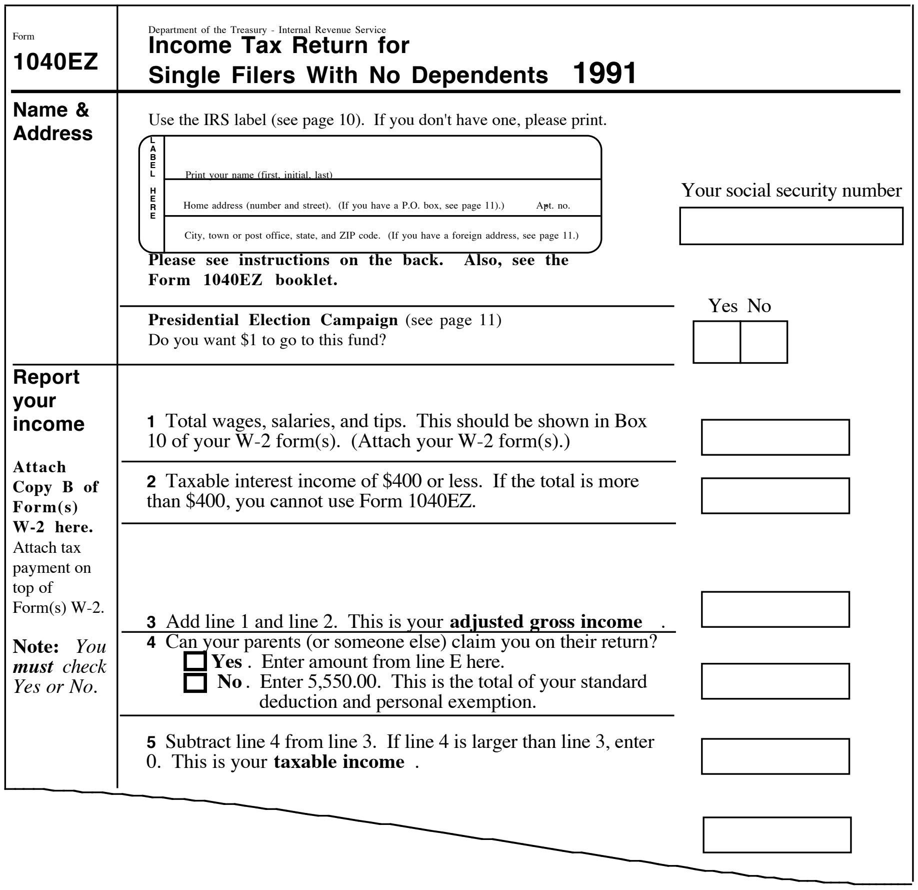
The WYSIWYT methodology has been integrated into the research language Forms/3. Here are three scenarios illustrating how it might look if integrated into Excel.

A-1. Scenario 1: An end user figures out and tests her income taxes

An end user has a printout of an income tax form from the U.S. Internal Revenue Service, such as in Figure A-1, in front of her, and she wants to use Excel to figure out the answers. To do this, she has created the spreadsheet in Figure A-2.

Although this spreadsheet is simple, there are several ways the user could end up reporting the wrong answer. Like many taxpayers, she may be struggling to gather all the required data, and may change her mind about the right data values to enter. If she has been taking shortcuts with the formulas, basing them on the conditions present in her first version of the data (such as not bothering to use a *max* operator in line 5 to prevent negatives), the formulas are probably not very general, and may cause problems if her data changes. For example, if she entered "line 4 - line 3" as the formula for line 5, but later changes line 4 to 5500 because her parents tell her they did not claim her this year after all, then the formula for line 5 will not give the correct answer. Similar problems could arise if she discovers that she entered data from the wrong box of her W–2 form into line 1, and so on.

Even in this simple case, the WYSIWYT methodology can help. Figure A-3 shows a mock-up of how it might be incorporated into Excel. All cells containing formulas (as opposed to data values) are initially red-bordered with checkboxes, as in Figure A-3 (top). The first time the user sees a red border, she moves her mouse over it and the tool tips inform her that "red borders mean untested and blue borders mean tested. You can check cells off when you approve of their values." The user checks off a value that she is sure is correct, and a checkmark (√) appears as in Figure A-3 (middle). Further, the border of this explicitly approved cell, as well as of cells contributing to it, becomes blue. If she then changed some data, any affected checkmarks would be replaced with blanks or question marks ("?") as described earlier in this paper, because the current value has not been checked off. But suppose that instead of replacing a data value, the user makes the formula change in line 4 alluded to above, changing the previous formula to the constant 5500 instead of the former reference to line E. Since the change she made involved a formula (the one she just changed to a data value), the affected cells' borders revert to red and downstream √s disappear, indicating that these cells are now completely untested again. (See Figure A-3 (bottom).) The maintenance of the "testedness" status of each cell throughout the editing process, as illustrated in Figure A-3 (bottom), is an important benefit of the approach. Without this feature, the user may not remember that her previous testing became irrelevant with her formula change, and now needs to be redone.



Form **1040EZ**

Department of the Treasury - Internal Revenue Service

Income Tax Return for Single Filers With No Dependents 1991

Name & Address

Use the IRS label (see page 10). If you don't have one, please print.

LABEL HERE

Print your name (first, initial, last)

Home address (number and street). (If you have a P.O. box, see page 11).) Apt. no.

City, town or post office, state, and ZIP code. (If you have a foreign address, see page 11.)

Please see instructions on the back. Also, see the Form 1040EZ booklet.

Your social security number

Presidential Election Campaign (see page 11)
Do you want $1 to go to this fund? Yes No

Report your income

Attach Copy B of Form(s) W-2 here. Attach tax payment on top of Form(s) W-2.

Note: *You **must** check Yes or No.*

1 Total wages, salaries, and tips. This should be shown in Box 10 of your W-2 form(s). (Attach your W-2 form(s).)

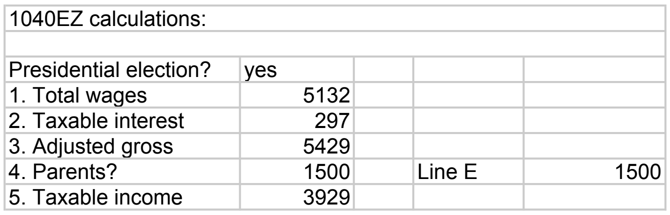
2 Taxable interest income of $400 or less. If the total is more than $400, you cannot use Form 1040EZ.

3 Add line 1 and line 2. This is your **adjusted gross income** .

4 Can your parents (or someone else) claim you on their return?
☐ **Yes** . Enter amount from line E here.
☐ **No** . Enter 5,550.00. This is the total of your standard deduction and personal exemption.

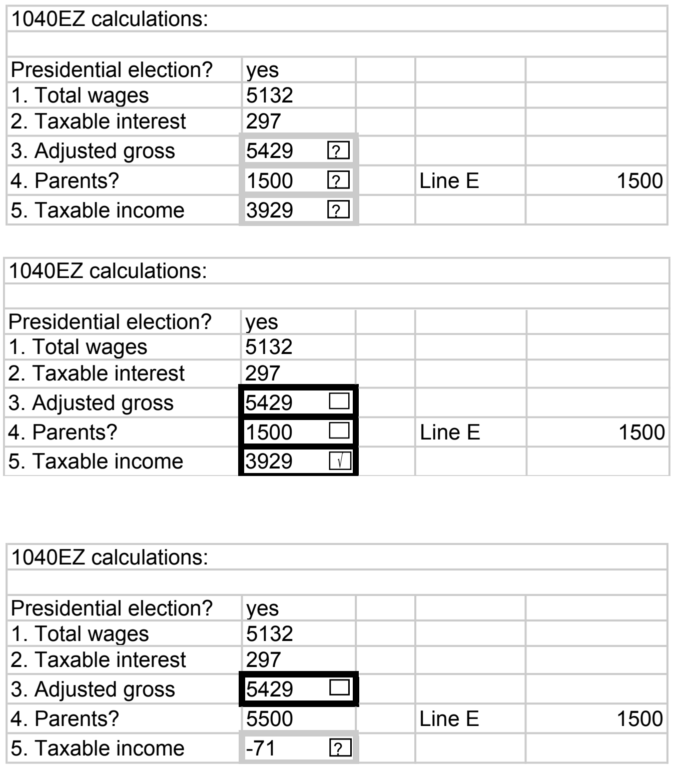
5 Subtract line 4 from line 3. If line 4 is larger than line 3, enter 0. This is your **taxable income** .

Figure A-1. A portion of a U.S. income tax form.



1040EZ calculations:				
Presidential election?	yes			
1. Total wages	5132			
2. Taxable interest	297			
3. Adjusted gross	5429			
4. Parents?	1500		Line E	1500
5. Taxable income	3929			

Figure A-2. The user's Excel spreadsheet to figure out the taxes. The first few cells are simply data values. Line 3's formula is "line 1 + line 2", line 4's formula is a reference to line E, and line 5's formula is "line 3 - line 4".



1040EZ calculations:				
Presidential election?	yes			
1. Total wages	5132			
2. Taxable interest	297			
3. Adjusted gross	5429 ?			
4. Parents?	1500 ?		Line E	1500
5. Taxable income	3929 ?			

1040EZ calculations:				
Presidential election?	yes			
1. Total wages	5132			
2. Taxable interest	297			
3. Adjusted gross	5429 ☐			
4. Parents?	1500 ☐		Line E	1500
5. Taxable income	3929 √			

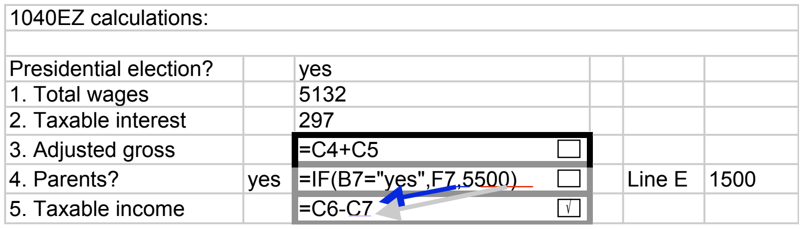
1040EZ calculations:				
Presidential election?	yes			
1. Total wages	5132			
2. Taxable interest	297			
3. Adjusted gross	5429 ☐			
4. Parents?	5500		Line E	1500
5. Taxable income	-71 ?			

FigureA-3. A mock-up of Excel if enhanced by the WYSIWYT technology: (Top): All cells containing formulas are initially red, meaning untested. (Middle): Whenever the user makes a decision that some data value is correct, she checks it off. The checkmark appears in the cell she explicitly validated, and all the borders of cells contributing to that correct value become more tested (closer to pure blue, shown as black in this picture). This example has such simple formulas, only the two colors red (light gray) and blue (black) are needed. (Bottom): The user changes the formula in line 4 to a constant. This change causes affected cells to be considered untested again.

A-2. Scenario 2: The user tests her income tax spreadsheet as she makes it more reusable

The next year, the user may want to improve the spreadsheet so that she can use it year after year without having to redesign each formula in the context of the current year's data values. For example, she adds the yes/no box from the IRS form's line 4 to her spreadsheet's line 4 and uses the *if* operator in the formula for line 4. Because of this *if*, she will need to try at least two test cases for line 4's cell to be considered tested: one that exercises the "*yes*" case and one that exercises the "*no*" case.

Because of this, when the user checks off one data value as in Figure A-4, the borders for lines 4 and 5 turn purple (50% blue and 50% red). To figure out how to make the purple cells turn blue, the user selects one of them and hits a "show details" key. The system then draws arrows pertaining to the subexpression relationships, with colors depicting which cases still need to be tested. The arrow from the last subexpression is red, telling the user that the "no" case still needs to be tried.



1040EZ calculations:					
Presidential election?		yes			
1. Total wages		5132			
2. Taxable interest		297			
3. Adjusted gross		=C4+C5 ☐			
4. Parents?	yes	=IF(B7="yes",F7,5500) ☐		Line E	1500
5. Taxable income		=C6-C7 ✓			

Figure A-4. Some cells require more than one test value to become completely tested, as this formula view with purple (medium gray) cell borders and red (light gray) and blue (black) arrows between subexpressions shows.

A-3. Scenario 3: A template developer tests an income tax spreadsheet for sale

It is well documented that many production spreadsheets contain bugs. To help address this problem, a developer with a full suite of income tax spreadsheet templates for sale could use the methodology to achieve organized test coverage of these income tax spreadsheets. This would not only be valuable when first developing the spreadsheets, but also in making sure that each formula change in subsequent years' revisions had been entered and tested.