End-User Software Engineering with Assertions

Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet
Oregon State University, Corvallis, Oregon, 97331 USA
{burnett, cook, pendse, grother, summet}@cs.orst.edu

Abstract

There has been little research on end-user program development beyond the activity of programming. Devising ways to address additional activities related to end-user program development may be critical, however, because research shows that a large proportion of the programs written by end users contain faults. Toward this end, we have been working on ways to provide formal "software engineering" methodologies to end-user programmers. This paper describes an approach we have developed for supporting assertions in end-user software, focusing on the spreadsheet paradigm. We also report the results of a controlled experiment, with 59 end-user subjects, to investigate the usefulness of this approach. Our results show that the end users were able to use the assertions to reason about their spreadsheets, and that doing so was tied to both greater correctness and greater efficiency.

1. Introduction

End-user programming has become a widespread phenomenon. For example, end users create and modify spreadsheets, they author web pages with links and complex formatting specifications, and they create macros and scripts. Although some of these programs are small, one-shot calculations, many are much more serious, affecting significant financial decisions and business transactions. Two recent NSF workshops have determined that end-user programming is in need of serious attention [5]. The reasons are compelling. The number of end-user programmers in the U.S. alone is expected to reach 55 million by 2005, as compared to only 2.75 million professional programmers [5]. Further, evidence abounds of the pervasiveness of errors in the software that end users create [20], with significant economic impact. In one single example, a Texas oil and gas firm lost millions of dollars through spreadsheet errors [19].

To address this problem, we have been researching methods for helping end users create more reliable software. Our approach draws aspects of theoretically sound methodologies, explored previously within the professional software engineering community, into the environments end users use to create software. We term the concept "end-user software engineering"; however, we do not claim or attempt to turn end users into software engineers or ask them learn traditional software engineering methodologies. Instead, the methodologies we employ aim to be not only accessible, but also verifiably productive for end users with little or no formal training in software engineering.

Assertions in the form of preconditions, postconditions, and invariants provide professional programmers with increased ability to maintain and debug software. Assertions provide attractive opportunities for adding rigor to end-user programming for two reasons. First, the use of assertions does not demand an "all or nothing" approach; it is possible to enter one or two assertions and gain some value, without committing to entering a purportedly complete set of assertions. Second, assertions provide a way to make explicit a user's mental model underlying a program, essentially, integrating "specifications" with that program. Such specifications, integrated into end-user programs, could tap into a host of opportunities for harnessing other software engineering methodologies, such as test suite improvement, specification-based test generation, automated test oracle generation, or proofs of program properties.

We have developed an approach for supporting assertions as a critical underpinning of our end-user software engineering work. We have prototyped our approach in the spreadsheet paradigm. Our assertions provide pre- and post-condition Boolean expressions about the results of cell formula execution. The approach supports a variety of relations, as well as composition via logical "and" and "or," in two concrete syntaxes. It is our hope that assertions will enable the user to identify and correct faulty formulas.

In this paper, we first present our approach to integrating assertions into an environment whose goal is to support end-user software engineering. We then use this environment to empirically explore the fundamental issues of whether end users can use assertions in the context of an end-user software engineering task, and whether and how their doing so will help them improve the correctness of their programs.

2. Assertions for end users

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, but unfortunately these formulas may contain inconsistencies or faults. 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 communicate other properties. Traditionally, assertions in the form of preconditions, postconditions, and invariants have fulfilled this need for professional programmers, providing 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. Our approach attempts to provide these same advantages to end-user programmers.

For professional software developers, the only widely used language that natively supports assertions is Eiffel

[15]. To allow programs in other languages to share at least some of the benefits of assertions, methods to add support for assertions to languages such as C, C++ and Awk have been developed (e.g., [2, 29]). Applications of such assertions to software engineering problems have proven promising. For example, there has been research on deriving runtime consistency checks for Ada programs [23, 25]. Rosenblum has shown that these assertions can be effective at detecting runtime errors [22]. However, traditional approaches that employ assertions are aimed at professional programmers, and are not geared toward end users.

2.1 An abstract syntax

As in the above approaches, 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. We also plan to eventually support inequality (≠) as a unary relation. This operator would be solely for convenience; it would not add power given the relations already present, since ≠ can be expressed as an or-assertion composing < with >. This abstract syntax follows CNF (Conjunctive Normal Form): each cell's collection of and-assertions, which in turn compose or-assertions, is intended to evaluate to true, and hence a spreadsheet's assertion is simply an "and" composition of all cells' assertions.

2.2 Concrete syntaxes for end users

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.

The graphical concrete syntax, depicted in Figure 1, 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.

The textual concrete syntax, depicted in Figure 2, 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 2. 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 the 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 [18].

The textual concrete syntax we present here bears some resemblance to recent work on English-like notations for formal specifications. Although assertions and other forms

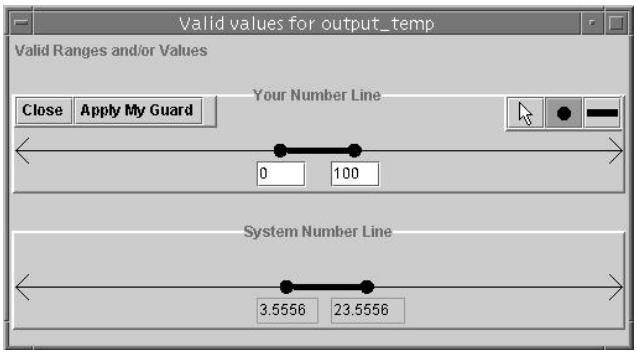

Figure 1: Two (conflicting) assertions "and"ed on the same cell.

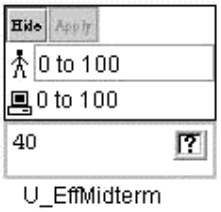

Figure 2: Two assertions in the textual syntax.

of specification such as property patterns [9] are usually presented using rigorous mathematical notations or finite state automata, there is work on expressing specifications in more accessible notations (e.g., [16, 26]). For example, Propel [26] is a multiple-view approach to properties that includes an English-like representation.

A central difference between that work and ours is that their properties and English-like syntax emphasize sequence. Our assertions have no temporal operators and cannot express sequence. This is appropriate because reasoning about sequence is not a programmer responsibility in declarative paradigms that focus on data definition, such as the spreadsheet paradigm, in which sequence is automatically derived from data dependencies.

2.3 Assertion sources' potential impact

We currently support two sources of assertions. *User assertions* are assertions that the user enters explicitly, one of which appears in Figure 2 next to the stick figure. *System-generated* assertions are assertions resulting from propagating assertions through formulas in the direction of dataflow (using straightforward logic and interval arithmetic), one of which appears in the figure next to the computer icon. (Details of propagation are described in [27].)

Other researchers have developed methods for generating program assertions without requiring programmer input as a source. Daikon [10] generates invariant assertions by extensive examination of a program's behavior over a large test suite. DIDUCE [12] deduces invariant assertions and uses them to check program correctness. DIDUCE has a training phase, in which it considers all behaviors correct and relaxes invariants to encompass them, and then a checking phase, which reports violations to the invariants inferred in the training phase. Raz et al.'s approach to semantic anomaly detection [21] uses off-the-shelf unsupervised learning and statistical techniques, including a variant of Daikon, to infer invariants about incoming data arriving from online data feeds, and their empirical work shows effectiveness. Recent work that can be described as inferring assertions related to correctness of end-user programs involves automatic detection of errors through outlier analysis [17]. This approach, which is similar in principle to that of Raz et al., has been developed in the domain of programming-by-demonstration for text processing.

An advantage of these researchers' inferential approaches is that they can relieve the programmer of having to conjure up and explicitly insert assertions. A disadvantage is that the assertions must be "guessed," and some of the guesses can be incorrect. The impact of incorrect guesses on end users' trust and willingness to work with assertions is an issue that requires exploration. We have not yet explored this issue in our research.

Because of multiple sources of assertions (currently two, potentially more), our approach provides the following three ways for assertions to potentially help users detect faults: (1) there might be multiple assertions on the cell that do not agree, termed an *assertion conflict*, (2) the value in the cell might not satisfy the cell's assertion(s), termed a *value violation*, or (3) inspection of a system-generated assertion might "look wrong" to the user. When any of these events occur, there may be either a fault in the program (spreadsheet formulas) or an error in the assertions.

2.4 Assertions in an end-user environment

An important difference between our approach to assertions and more traditional approaches is that ours is a component of an integrated set of software engineering features designed particularly for end users. It is not a separate tool, and a user need not use it in an "all or nothing" manner. Instead, the user might enter a few assertions, immediately see their effects on the formulas entered so far, and then apply other forms of validation on other formulas. Feedback about assertions is integrated with the variables (cells) to which they apply. For example, the assertions about cell U_EffMidterm appear at the top of that cell, as already shown in Figure 2. If the two assertions about cell U_EffMidterm were in conflict, their icons would be circled in red. If the cell's value did not satisfy its assertions, the value would be circled.

Microsoft Excel, a popular commercial spreadsheet application, has a data validation feature integrated into the environment that bears a surface-level similarity to the assertions in our environment. Excel, however, does not support propagation of assertions to other cells, does not automatically display assertions, and does not maintain the display of assertion violations when changes are made. That is, the data validation feature is primarily an optional data entry check that can be invoked from time to time by the user. This is quite different from our approach, because our assertions combine into a network that is the basis of an ever-present reasoning system that gives continuous and up-to-date visual feedback.

In addition to the integration of assertion feedback with the program source code (formulas) and values, in our prototype environment there is also testing feedback integrated in the same fine-grained way via our already incorporated spreadsheet testing methodology known as WYSIWYT ("What You See is What You Test") [11, 24]. The WYSIWYT methodology implements a dataflow test adequacy criterion. The criterion for complete "testedness" under this methodology is that each executable definition-use (du) association in the spreadsheet be exercised by test data in such a way that the du-association contributes (directly or indirectly) to the display of a value that is subsequently pronounced correct (validated) by the user. Users can convey to the system that a cell's value is correct for the spreadsheet's inputs by checking the checkbox in the upper right corner of that cell. This results in a change of cell border color, which is used to represent the testedness status of a cell. Red means untested, shades of purple mean partially tested, and blue means fully tested. There is also a "percent tested" indicator at the top of the spreadsheet that displays information about the spreadsheet as a whole. These devices are always kept up-to-date. For example, when a formula is entered or modified, the cell border turns red because the cell is untested; borders of cells that reference the modified cell also turn red. Because a principle of our end-user software engineering approach is that the assertions be integrated with testing support, the investigation of assertions in this paper is in the context of WYSIWYT.

2.5 Example

We close this section by presenting a simple illustration of our prototype assertion mechanism in the context of the Forms/3 research spreadsheet language [6]. Figure 3(a) shows a portion of a Forms/3 spreadsheet which 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 it 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 [28]. The quotes are from a recording of the subject's commentary.

First, the user changed the assertion on the input_cell to range from 0 to 100. This caused several red violation ovals to appear, as in Figure 3(b), because the values in input_cell, a, b, and output_cell were now out of range and the assertion on output_cell 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_cell from 200 to 75 ("something between zero and 100"), and then set the formula in cell a to "input_cell * 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 the output_cell (which still had the formula "b / 9"). The user now chose to deal with the assertion conflict on the output_cell, and clicked on the guard icon to view the details in the graphical syntax.

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_cell's formula, making it a reference to cell b. This resulted in the value of output_cell 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 shown in Figure 3(c).

3. Experiment

Our initial think-aloud study provided early insights into five end users' use and understanding of assertions, but did not statistically evaluate effectiveness. Thus, to investigate empirically whether and how this approach to assertions would increase users' effectiveness at eliminating faults, we conducted a controlled experiment involving 59 subjects to investigate the following research questions:

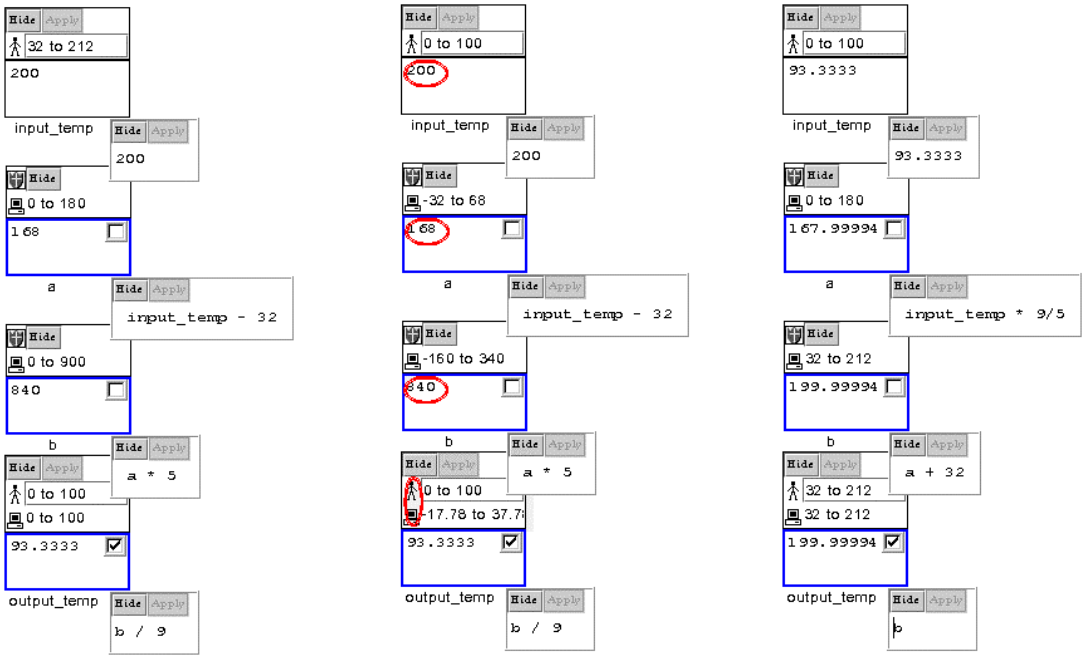

Figure 3: Example at three points in the modification task.

Will assertions users be more effective debuggers?
Will users understand assertions?
Will assertions help users judge the correctness of their programs (spreadsheets)?

We decided to isolate as much as possible the assertion effects (dependent) variables from that of choosing whether to enter assertions, which is also a dependent variable. The effectiveness results could have been confounded if we also required subjects to choose to master assertion entry. By largely eliminating this choice as a factor, we gained high assurance that assertions would be present in the tasks, which is what enabled us to measure effectiveness and other effects of assertions. We return to this point in Section 5.

3.1 Procedures

The experiment was conducted in a Windows computer lab. The subjects were seated one per computer. Prior to running the experiment, we conducted a four-subject pilot study to test the experimental procedures and materials.

At the beginning of the experiment, the subjects were asked to fill out a background questionnaire. The session continued with a 35-minute tutorial to familiarize subjects with the environment, in which they worked on two spreadsheets along with the instructor. Subjects worked in identical end-user software engineering environments, except that the Treatment group (two sessions) had the assertions feature in their environment whereas the Control group (two sessions) did not. To reduce memorization skills as a factor, a quick-reference sheet listing the environment's features was provided to the subjects. The sheet remained with them throughout the experiment, and they were allowed to take notes on it. To ensure that both groups had equal time familiarizing themselves with their environments, we balanced the time spent presenting assertions to the Treatment group by providing additional testing practice time to the Control subjects.

After the tutorial, subjects began their experimental tasks of debugging a grades spreadsheet (20 minutes) and a payroll spreadsheet (15 minutes). The use of two spreadsheets reduced the chances of obscuring the results because of any one spreadsheet's particular characteristics. The tasks necessarily involved time limits, to make sure subjects would work on both spreadsheets, and to remove possible peer influence of some subjects leaving early. The time limits were drawn from times we observed to be required by the pilot study's subjects. The experiment was counterbalanced with respect to problem to prevent learning effects from affecting the results: all subjects worked both problems, but half of each group debugged the problems in the opposite order.

Initially, no assertions were on display for either problem. The Treatment subjects' scenario was that a previous user had devised assertions for some of the cells, which subjects could use by clicking that cell's "see user guard" button. Clicking this button had the same effect as if a helpful user had immediately typed in a user assertion for that cell, but eliminated issues of users having to learn how to enter assertions, or choosing to go to that much trouble. We decided on this procedure, instead of having the assertions already present, because the approach is for an incremental, interactive environment, in which assertions are likely to be entered and dealt with incrementally. Other than the button, no other assertion editing devices were available. Once an assertion had been entered via the button, it was propagated and displayed as described in Section 2.

At the end of each task, subjects were given a post-task questionnaire which included questions assessing their comprehension and attitudes about the features they had used, and also questions in which they self-rated their performance. Besides the questionnaire responses, data included the subjects' actions, which were electronically captured in transcript files, and their final spreadsheets.

3.2 Subjects

The subjects were non-computer science students who had little or no programming experience. The 59 subjects were randomly divided into two groups: 30 in the Treatment group and 29 in the Control group. Of the 59 subjects, 23 were business students, 22 came from a wide range of sciences (such as psychology, biology, geography, pharmacy, and animal science), and the remaining 14 came from a variety of other non-computer science majors. The average (self-reported) GPA of the Control group was lower than that of the Treatment group but there was no statistically significant difference. Roughly 60% of the subjects had a little programming experience, due to the fact that it is common these days for business and science students to have taken a high school or college class in programming. Statistical tests on subject data showed no significant differences between the Treatment group and Control group in any of these attributes.

3.3 Problems

The debugging problems were Grades (Figure 4) and Weekly Pay (not shown). Subjects were given a written description of each spreadsheet. The description explained the functionality of the spreadsheet, and included the ranges of valid values for some of the cells, so that all subjects would have exactly the same information about the spreadsheets, regardless of whether their environment included assertions. The subjects were instructed to "test the spreadsheet thoroughly to ensure that it does not contain errors and works according to the spreadsheet description. Also if you encounter any errors in the spreadsheet, fix them."

The Grades spreadsheet (24 cells) calculates the grades of two students, one graduate and the other undergraduate, with different grading criteria. There are 11 faults in seven cells of this spreadsheet. The Grades spreadsheet was laid out such that it required scrolling, to ensure that *not* all assertions and assertion conflicts would necessarily be visible at any one time. The Weekly Pay spreadsheet (19 cells) calculates the weekly pay and income tax withholding of a salesperson. There are seven faults in five cells of this spreadsheet.

To facilitate comparing the impact of assertions with whatever other mechanisms subjects might use to identify and correct faults, we devised assertions that would expose only some of the faults. To avoid biasing results against the

Control group, we included assertions only for cells whose ranges were explicitly stated on the problem description. In choosing which ranges to state, we included all ranges that could be clearly justified without a calculator. Via the button, all Grades cells had user assertions available, which could expose only 8 of the 11 faults. For Weekly Pay, user assertions able to expose 3 of the 7 faults were available.

3.4 Fault types

Allwood [1] classified faults in spreadsheets as mechanical, logical and omission faults, and this scheme is also used in Panko's work [20]. Under Allwood's categorization, mechanical faults were further classified as simple typographical errors or wrong cell references in the cell formulas. Mistakes in reasoning were classified as logical faults. Logical faults in spreadsheets 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 formulas.

We drew from this research by including faults from each category in each problem. However, the precise distinctions between logical and mechanical are not clear for some types of faults in end-user programming. For example, when computing an average, does dividing by the wrong number mean the subject typed it wrong, or that they are confused about computing averages? In our think-aloud studies we have collected data in which end-user subjects made exactly this error for both of these reasons. Thus, we combined the first two categories into one and then, to be sure coverage of both would be achieved, included several different subtypes under it: incorrect references (which Allwood would classify as mechanical), incorrect constants or an omitted character (could be either logical or mechanical), incorrect operators or application of operators (which Allwood would classify as logical), and an extra subexpression (logical). We also included faults from the third category, omission faults. Table 1 shows a summary of the spreadsheets' faults.

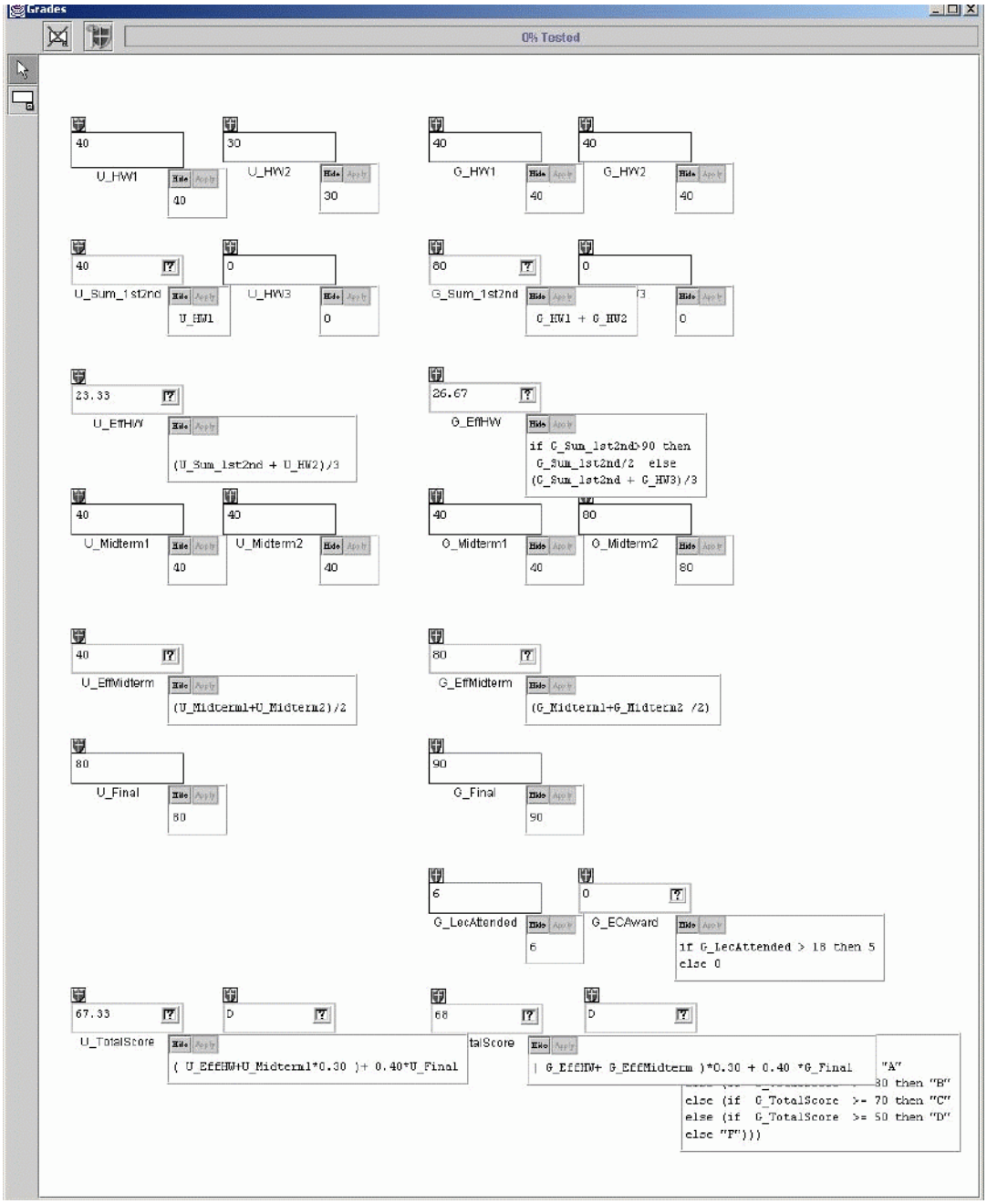

Figure 4: The Grades Spreadsheet.

Another classification scheme we have found to be useful in our previous research involves two fault types: reference faults, which are faults of incorrect or missing references, and non-reference faults, which are all other faults. In this study, the omissions and incorrect references are reference faults (7 in total), and the remaining types are non-

reference faults (11 in total).

The faults not exposed by assertions were distributed across all of Table 1's categories, and were also assigned in proportion to the reference/ non-reference categorization.

4. Results

4.1 Debugging effectiveness

4.1.1 Accuracy by subject group. We chose to analyze the act of identifying a fault separately from the act of correcting it. The reason was that there is little information in the literature about end users' abilities to correct a fault even if they have managed to identify it, a gap we hope to help fill. For cells containing exactly one fault, we defined a fault to have been *identified* if the subject changed the formula containing the fault and to have been *corrected* if the changes resulted in a correct formula. When a cell contained two faults, we partitioned the formula into two disjoint subexpressions, one containing each fault. Then if the particular fault's subexpression was changed (or changed correctly), the fault was defined to be identified (or corrected). For example, in cell U_EffHW the original formula in the cell is

(U_Sum1st2nd+U_HW2)/3

and the correct formula is

(U_Sum1st2nd+U_HW3)/2

The two faults are adding U_HW2 instead of U_HW3 and dividing by 3 instead of 2. The two parts to the formula are the sum and the division. Hence a subject would be credited with identifying (correcting) the fault in the sum if the sum subexpression was changed (changed correctly) and credited with identifying (correcting) the fault in the division if the divisor was changed (changed correctly).

We now state the following (null) hypotheses as statistical vehicles for investigating the accuracy question:

H1: Identify: There will be no difference between the Treatment group's and Control group's number of faults identified.

H2: Correctness: There will be no difference between the Treatment group's and Control group's number of faults corrected.

The faults identified and corrected are summarized in Table 2. A two-factor analysis of variance with repeated measures on the problem (Grades and Weekly Pay) and group (Control and Treatment) showed that the Treatment subjects both identified significantly more faults ($F=16.59$, $df=1$, $p=.0001$) and corrected significantly more faults ($F=11.74$, $df=1$, $p=.0011$) than did the Control subjects. Thus, hypotheses H1 and H2 must both be rejected.

Table 1: Classifications of seeded faults.

	Omission	Logical and Mechanical			
		Ref.	Const. or char.	Oper-ator	Extra subexpr
Grades	2	2	3	3	1
Weekly Pay	1	2	2	2	0

Both the Treatment and Control subjects corrected nearly equal percentages of the faults they identified (Table 3). That is, when they found the faults, they generally succeeded at correcting them. In fact, in the Grades problem the percents corrected once identified were exactly the same for subjects with assertions as for those without assertions, and the difference was very small in the Weekly Pay problem. This suggests that the assertion advantages in *identifying* faults were enough to produce the eventual advantages in correcting them.

4.1.2 Accuracy by fault type. Did assertions especially help with any particular type of faults? Fisher's Exact Test showed that the Treatment group identified significantly more instances on seven of the 18 faults, but they were not all of the same type. In fact, these faults covered four of the five types of faults we seeded (recall Table 1): three were operator or operator application faults, two were incorrect constants, one was an omission, and one was an incorrect reference. Thus, according to Allwood's classification scheme, assertions contributed to effectiveness across a wide range of faults.

Considering instead the reference/non-reference classification of faults reveals an interesting attribute of assertions' effectiveness: subjects in the Treatment group were significantly more effective at identifying and correcting non-reference faults in both problems (Grades corrected: $F=9.67$, $df=1$, $p=.0029$; Weekly Pay corrected: $F=8.26$, $df=1$, $p=.0057$). There was also significance for reference faults on one of the problems (Grades corrected: $F=4.67$, $df=1$, $p=.0348$; Weekly Pay corrected: $F=3.72$, $df=1$, $p=.0506$). In contrast to this, previous empirical work regarding the effectiveness of the WYSIWYT testing methodology (which is based on a dataflow adequacy criterion) on detecting faults has revealed that WYSIWYT's advantage has been primarily in identifying and correcting reference faults [7].

Since both groups' subjects had WYSIWYT available to them, which is strong on reference faults, it is not surprising that the reference fault improvement brought by assertions was significant on only one problem. The more interesting result is that assertions helped so significantly with non-reference faults, suggesting that the addition of assertions

Table 2: Mean number of faults identified and corrected.

	Identified	Corrected
Grades (11 faults)		
Control	5.69	4.83
Treatment	8.07	6.83
Weekly Pay (7 faults)		
Control	4.97	4.59
Treatment	5.90	5.70

Table 3: Percent of identified faults that were eventually corrected.

	Grades	Weekly Pay
Control (n=29)	84.8%	92.4%
Treatment (n=30)	84.8%	96.6%

into the environment fills a need not met effectively by the dataflow testing methodology alone.

4.1.3 Debugging speed

H3: Speed: There will be no difference between the Treatment group's and Control group's speed in correcting the faults.

We partitioned the task time into 5-minute blocks—the first 5 minutes, second 5 minutes, and so on—and counted the number of faults that had been corrected by the end of each partition. Since subjects were given more time on Grades than on Weekly Pay, there are four partitions for Grades and three for Weekly Pay. Each block in the stacks in Figure 5 covers a 5-minute time period (first 5-minute period at the bottom, etc.), and the data value in it is the number of faults identified/corrected during that time period. As the figure shows, the Treatment subjects identified and corrected faults faster in each of the 5-minute blocks. The Treatment subjects' advantage began very early. The correctness differences were significant for the Weekly Pay problem at the 5-minute mark (identified F=3.70, df=1, p=.0595; corrected F=4.19, df=1, p=.0453), and the subjects were significantly more effective at identifying and correcting faults during the first 5 minutes for the Grades problem (identified F=13.04, df=1, p=.0006; corrected F=15.67, df=1, p=.0002). Thus, H3 is rejected.

4.2 Did the users understand assertions?

Assertions had significant positive impacts on subjects' abilities to remove the faults, but to what extent did they understand what the assertions meant? At least some understanding is important, because research in on-line trust has shown that users must believe they understand the system's reasoning, at least roughly, in order to trust its results enough to use them effectively [4, 8].

To help assess the subjects' understanding, we asked several questions about the meanings of the different devices in the context of a two-cell example spreadsheet. The results, which are summarized in Table 4, imply that the Treatment subjects understood the meaning of the assertion features reasonably well. The last question, in which they were required to fill in the result of propagating cellA's assertion (0 to 5) through cellB's formula (cellA + 10), was the most difficult for them, but 53% answered it correctly (10 to 15). An additional 23% answered "0 to 15." The fact that this additional 23% were correct on the upper limit seems to indicate that they had a rough, but imperfect, idea of how propagation worked.

One strong indicator that the subjects believed their understanding was "good enough" for reliance upon assertions can be found in their ratings. Subjects rated the helpfulness of three assertion features and also two testing features. Treatment subjects rated the assertion conflicts the most helpful of all the features. On a scale of 1 to 5 (not helpful to very helpful), their ratings for the assertion conflicts averaged 4.4 (median 5). See Table 5.

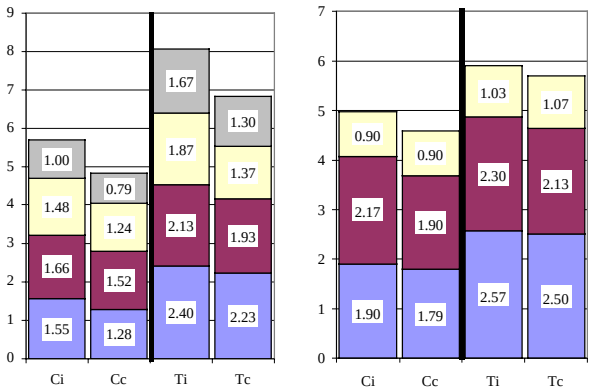

Figure 5: Average faults identified by each group (Ci, Ti) and corrected (Cc, Tc). Left graph: Grades. Right graph: Weekly Pay.

4.3 Judging correctness

After the subjects had completed the experimental tasks, we asked them to what extent they believed they had managed to identify and to correct all the faults. In the practice of software development, it is often this question that the developer uses to decide whether the software is ready to use. For this reason, helping users make reasonable judgments in answer to this question can be important in preventing software from going into use prematurely.

The questionnaire asked them to rate on a 1 ("not confident") to 5 ("very confident") scale, for each problem, how confident they were that they had identified and corrected all the faults. The issue relevant to the effectiveness of assertions is to what extent these self-ratings were correlated with correctness. To this end, we compared self-ratings to actual performance.

Table 4: Percentage of correct responses given by Treatment subjects. (Correct responses are shown in parentheses.)

Question	% correct
What does the red oval on cellA mean? (The value is outside the valid range.)	93%
What does the little stick figure in the cellA guard mean? (The user supplied the guard.)	83%
Why is there a stick figure and a computer on cellB's guard? (The guard was supplied by both Forms/3 and by the user.)	87%
What does the red oval on the cellB guard mean? (The user and Forms/3 disagree on the valid range(s) for this cell.)	63%
Given the formula in cellB and the guard on cellA, what do you think Forms/3 says are the valid range(s) for cellB? (10 to 15)	53%

Table 5: Subjects' helpfulness ratings.

	Assertion conflicts	"Tested" border colors	System-generated assertions	User assertions	% tested
Control	N/A	4.2	N/A	N/A	2.9
Treatment	4.4	4.3	4.2	4.0	3.6

H4: Self ratings of correctness: There will be no difference between Control versus Treatment subjects' self-ratings as predictors of correctness (number of faults in the spreadsheets at the end of the experiment).

Regression analysis is the appropriate test for a predictive question of this type. The results for correcting the faults are shown in Table 6; the results for identifying the faults (not shown) are similar. The regression coefficient is the slope of the least squares fitting of the ratings against the faults that were corrected. As the table shows, the Treatment group gave self-ratings that were statistically significant predictors of actual performance, with regression coefficients that were significantly different from zero for both problems. (This was true also for identifying the faults.) The Control group's self-ratings, on the other hand, were ineffective as predictors, and their regression coefficients were not significantly different from zero. In fact, for one problem, the regression coefficient is slightly negative, indicating that the Control group's predictions had a slight tendency to be the *opposite* of their actual performance. Clearly, H4 is rejected.

This result is important. It is a well known and robust result from behavioral science that humans are overconfident about the work they do, and that this tendency is extremely resilient. In the spreadsheet literature this tendency has become known as overconfidence. Such overconfidence has been widely reported in many spreadsheet studies (surveyed in [20]), in the Forms/3 environment [30], and in studies of software professionals [14]. Although overconfidence of spreadsheet accuracy has previously been reduced to some extent by the WYSIWYT testing methodology, the reductions have not been sufficient to correlate in a statistically significant way with accuracy [13]. The improved level of judgment associated with assertions thus fills a gap in end-user programming, which could prevent some of their software from going into production use too early.

5. Discussion: Will users enter assertions?

We have explained that we designed our experiment to isolate effectiveness of assertions when they were present, and thus while subjects had a choice as to whether and which assertions were present, they did not have to actually enter them. Therefore, in a follow-up pilot experiment, we focused specifically on the question of users choosing to enter assertions via the concrete syntaxes of Section 2. In the follow-up pilot, assertions were not explained or even mentioned in the tutorial. Instead, the tutorial instructed users to explore the environment in any way they wished, and gave them time to practice doing so. In their work on the same problems as for the experiment reported here, 25 of the 30 subjects (83%) did choose to enter assertions, and once they entered one they entered more, averaging 18 assertions per subject. Further, 96% of the assertions they entered were the same as assertions we provided in the experiment reported here.

Table 6: Self-ratings as correctness predictors.

	Regression coeff.	t	df	Significance
Grades				
Control	0.168	1.434	27	p = .1645
Treatment	0.210	2.188	28	p = .0375
Weekly Pay				
Control	-0.038	-0.261	27	p = .7963
Treatment	0.477	2.165	28	p = .0394

6. Threats to validity

We attempted to address threats to internal validity for this study by randomly distributing subjects among the groups and statistically checking the distribution for significant differences, by including two problems and counterbalancing them, by distributing the seeded faults among a variety of fault types, by equalizing training time, and by selecting problems from familiar domains.

As in most controlled experiments, however, threats to external validity are more difficult to address given the need to control all other factors. For example, the spreadsheets used may seem rather simple, but most subjects did not achieve 100% correctness of their formulas, indicating that the spreadsheets were not too simple for the amount of time given. Also, although the prototype and experiment included only constants as assertion operands, we do not view this as a threat to validity, because the assertions are fairly powerful even with this restriction. However, the faults we used to seed the spreadsheets may not have been representative of formula faults in real-world spreadsheets.

The fact that the experiment included explicit time limits is a threat to external validity. Explicit time limits were necessary to eliminate the internal threat of subjects stopping only because other subjects were finishing and leaving. In the real world, the amount of time available for debugging is constrained by time pressures, but explicit time limits do not ideally simulate these time pressures.

The short tutorial prior to the experiment included an informal explanation of how system-generated assertions are created through propagation, but no such explanation would be likely in the real world. We are working on including the essential information about assertions in a network of context-sensitive explanations [3]. The pilot study we mentioned in the previous section included an early version of such a system. The pilot subjects seemed to understand assertions enough to use them effectively, but further empirical work after the explanation system is complete is required to follow up on this issue.

7. Conclusion

We have presented an approach for supporting assertions in end-user software, focusing on the spreadsheet paradigm. Our assertions provide pre- and postcondition expressions about the results of cell executions, and can be generated by the user or the system. The concrete syntaxes by which assertions are represented look to users like simple points and ranges, but these syntaxes are sufficient to express an entire abstract assertion syntax of substantial power.

To evaluate our approach empirically, we conducted a controlled experiment with 59 end-user subjects. The most important results were:

- Assertions did indeed help end users debug more effectively and more efficiently.
- The effectiveness boost applied across a wide range of fault types. Moreover, assertions were extremely effective with non-reference faults, a class that had not been amenable to detection by our dataflow testing methodology for end users.
- Assertions clearly combated the well established tendency of end users toward overconfidence, by significantly improving their ability to judge whether they had done enough to ensure the correctness of their spreadsheets.

Perhaps the most surprising result of all is that end users not only understood assertions, they actually liked them. They rated assertion conflicts as being more helpful than any other feature, and in a follow-up pilot, if subjects discovered assertions, they chose to enter quite a few of them. The facts that they understood and liked assertions are critical outcomes because of their importance in determining whether end users will ultimately use assertions in the real world.

Acknowledgments

We thank the members of the Visual Programming Research Group for their feedback and help. This work was supported in part by NSF under ITR-0082265.

References

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

[2] Auguston, M., Banerjee, S., Mamnani, M., Nabi, G., Reinfelds, J., Sarkans, U., and Strnad, I. A debugger and assertion checker for the Awk programming language. *Int. Conf. Soft. Eng.*, 1996.

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

[4] Belkin, N. Helping people find what they don't know, *Comm. ACM* 41(8), Aug. 2000, 58-61

[5] Boehm, B. and Basili,, V. Gaining intellectual control of software development, *Computer* 33(5), May 2000, 27-33.

[6] Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., Yang, S. Forms/3: a first-order visual language to explore the boundaries of the spreadsheet paradigm, *J. Functional Programming*, Mar. 2001, 155-206.

[7] Cook, C., Rothermel, K., Burnett, M., Adams, T., Rothermel, G., Sheretov, A., Cort, F., Reichwein, J. Does a visual "testedness" methodology aid debugging? TR #99-60-07, Oregon State Univ., rev. March 2001.

[8] Corritore, C., Kracher, B., and Wiedenbeck, S. Trust in the online environment, *HCI International, Vol. 1*, New Orleans, LA, Aug. 2001, 1548-1552.

[9] Dwyer, M., Avrunin, G., and Corbett, J. Patterns in property specifications for finite-state verification, *Int. Conf. Soft. Eng.*, Los Angeles, CA, May 1999, 411-420.

[10] Ernst, M., Cockrell, J., Griswold, W., and Notkin, D. Dynamically discovering likely program invariants to support program evolution, *Int. Conf. Soft. Eng.*, Los Angeles, CA, May 1999, 213-224.

[11] Fisher, M., Cao, M., Rothermel, G., Cook, C., and Burnett, M. Automated test case generation for spreadsheets, *Int. Conf. Soft. Eng.*, Orlando, FL, May 2002, 141-151.

[12] Hangal, S. and Lam, M. Tracking down software bugs using automatic anomaly detection, *Int. Conf. Soft. Eng.*, Orlando, FL, May 2002, 291-301.

[13] Krishna, V., Cook, C., Keller, D., Cantrell, J., Wallace, C., Burnett, M., Rothermel, G. Incorporating incremental validation and impact analysis into spreadsheet maintenance: an empirical study, *IEEE Int. Conf. Soft. Maintenance*, Florence, Italy, Nov. 2001, 72-81.

[14] Leventhal, L., Teasley, B., and Rohlman, D. Analyses of factors related to positive test bias in software testing. *Int. J. Human-Computer Studies* 41, 1994, 717-749.

[15] Meyer, B. Applying "Design by Contract," *Computer* 25(10), October 1992, 40-51.

[16] Michael, J., Ong, V., and Rowe, N. Natural-language processing support for developing policy-governed software systems, *Int. Conf. Technology for Object-Oriented Languages and Systems*, Santa Barbara, CA, 2001.

[17] Miller, R. and Myers, B. Outlier finding: focusing user attention on possible errors, *ACM Symp. User Interface Software and Technology*, Nov. 2001.

[18] Pane, J., Ratanamahatan, C., and Myers, B. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Human-Computer Studies* 54(2), Feb. 2001, 237-264.

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

[20] Panko, R. What we know about spreadsheet errors. *J. End User Computing*, Spring 1998, 15-21.

[21] Raz, O., Koopman, P., and Shaw, M. Semantic anomaly detection in online data sources, *Int. Conf. Soft. Eng.*, Orlando, FL, May 2002, 302-312.

[22] Rosenblum, D. A practical approach to programming with assertions, *IEEE Trans. Soft. Eng.*, Jan. 1995, 19-31.

[23] Rosenblum, D., Sankar, S. and Luckham, D. Concurrent runtime checking of Annotated Ada programs, *Conf. Foundations Soft. Technology and Theoretical Computer Science (LNCS 241)*. NY, Springer-Verlag, Dec. 1986, 10-35.

[24] Rothermel, G., Burnett, M., Li, L., DuPuis, C., Sheretov, A. A methodology for testing spreadsheets, *ACM Trans. Soft. Eng. and Methodology*, Jan. 2001, 110-147.

[25] Sankar, S., Mandal, M. Concurrent runtime monitoring of formally specified programs, *Computer*, Mar. 1993, 32-41.

[26] Smith, R., Avrunin, G., Clarke, L., Osterweil, L. Propel: an approach supporting property elucidation, *Int. Conf. Soft. Eng.*, Orlando, FL, May 2002, 11-21.

[27] Summet, J. and Burnett, M. End-user assertions: propagating their implications, TR 02-60-04, Oregon State Univ., 2002.

[28] Wallace, C., Cook, C., Summet, J., and Burnett, M. Assertions in end-user software engineering: a think-aloud study (Tech Note), *IEEE Symp. Human-Centric Lang. Environ.*, Arlington, VA, Sept. 2002, 63-65.

[29] Welch, D., String, S. An exception-based assertion mechanism for C++. *J. Obj. Oriented Prog.* 11(4), 1998, 50-60.

[30] Wilcox, E., Atwood, J., Burnett, M., Cadiz, J., and Cook, C. Does continuous visual feedback aid debugging in direct-manipulation programming systems? *ACM Conf. Human Factors in Computing Systems*, Mar. 1997, 258-265.