By Margaret Burnett, Curtis Cook, *and* Gregg Rothermel

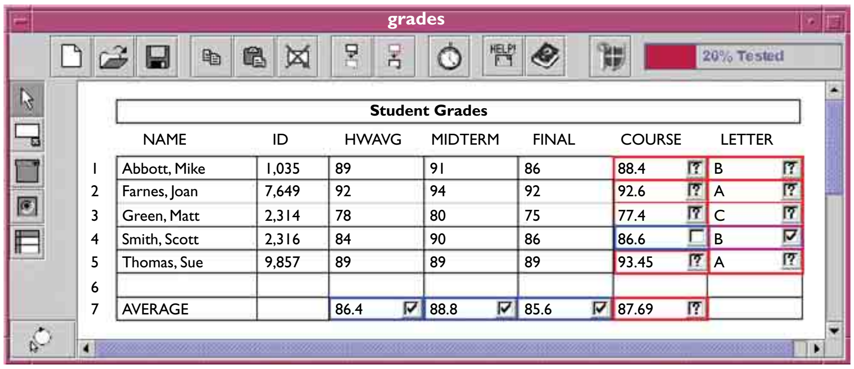
A strategy that allows end users the ability to perform quality control methods as well as inspires them to enhance the dependability of their software themselves.

END-USER SOFTWARE ENGINEERING

END-USER PROGRAMMING HAS BECOME THE MOST COMMON FORM OF PROgramming in use today [2], but there has been little investigation into the dependability of the programs end users create. This is problematic because the dependability of these programs can be very important; in some cases, errors in end-user programs, such as formula errors in spreadsheets, have cost millions of dollars. (For example, see www.theregister.co.uk/content/67/31298.html or panko.cba.hawaii.edu/ssr/Mypapers/whatknow.htm.) We have been investigating ways to address this problem by developing a software engineering paradigm viable for end-user programming, an approach we call *end-user software engineering.*

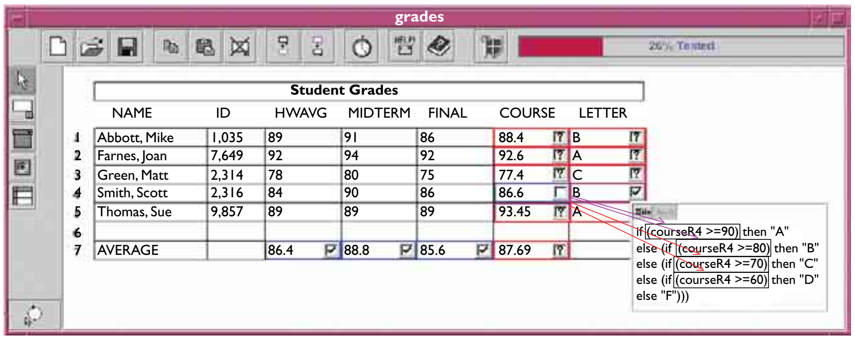
End-user software engineering does not mimic the traditional approaches of segregated support for each element of the software engineering life cycle, nor does it ask the user to think in those terms. Instead, it employs a feedback loop supported by behind-the-scenes reasoning, with which the system and user collaborate to monitor dependability as the end user's program evolves. This approach helps guard against the introduction of faults[1] in the user's program and if faults have already been introduced, helps the user detect and locate them. Because spreadsheet languages are the most widely used end-user programming languages to date—in fact, they may be the most widely used of *all* programming languages—we have prototyped

[1]We follow the standard terminology for discussing program errors. A "failure" is an incorrect output, and a "fault" is the incorrect element(s) of source code causing the failure. For example, an answer of "D" in a spreadsheet cell if the student's grade should actually be a "B" is a failure; the incorrect formula, such as omission of one of the student's test grades in the sum upon which his/her letter grade is based, is the fault.



	Student Grades						
	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4 ?	B ?
2	Farnes, Joan	7,649	92	94	92	92.6 ?	A ?
3	Green, Matt	2,314	78	80	75	77.4 ?	C ?
4	Smith, Scott	2,316	84	90	86	86.6 ☐	B ☑
5	Thomas, Sue	9,857	89	89	89	93.45 ?	A ?
6							
7	AVERAGE		86.4 ☑	88.8 ☑	85.6 ☑	87.69 ?	

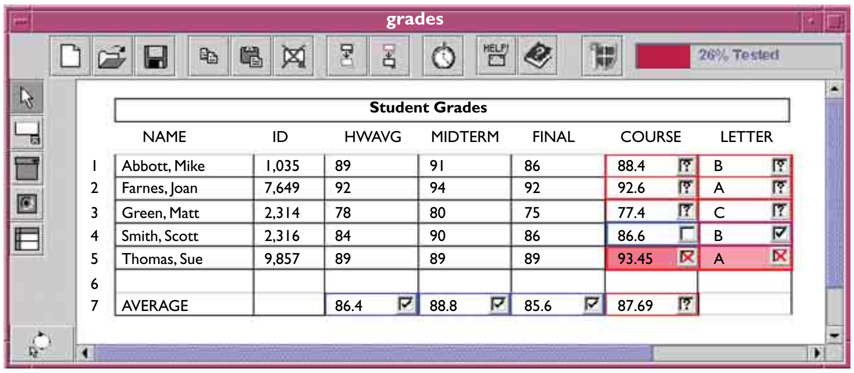
(a)



	Student Grades						
	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4 ?	B ?
2	Farnes, Joan	7,649	92	94	92	92.6 ?	A ?
3	Green, Matt	2,314	78	80	75	77.4 ?	C ?
4	Smith, Scott	2,316	84	90	86	86.6 ☐	B ☑
5	Thomas, Sue	9,857	89	89	89	93.45 ?	A
6							
7	AVERAGE		86.4 ☑	88.8 ☑	85.6 ☑	87.69 ?	

```
if (courseR4 >=90) then "A"
else (if (courseR4 >=80) then "B"
else (if (courseR4 >=70) then "C"
else (if (courseR4 >=60) then "D"
else "F")))
```

(b)



	Student Grades						
	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4 ?	B ?
2	Farnes, Joan	7,649	92	94	92	92.6 ?	A ?
3	Green, Matt	2,314	78	80	75	77.4 ?	C ?
4	Smith, Scott	2,316	84	90	86	86.6 ☐	B ☑
5	Thomas, Sue	9,857	89	89	89	93.45 ☒	A ☒
6							
7	AVERAGE		86.4 ☑	88.8 ☑	85.6 ☑	87.69 ?	

(c)

Figure 1. The teacher's sequence of interactions with WYSIWYT testing.

our approach in the spreadsheet paradigm. Our prototype includes the following end-user software engineering devices:

- An interactive testing methodology to help end-user programmers test;
- Fault localization capabilities to help users find the faults that testing may have revealed;
- Interactive assertions to continually monitor values the program produces, and alert users to potential discrepancies; and
- Motivational devices that gently attempt to interest end users in appropriate software engineering behaviors at suitable moments.

In this article, we describe how these devices can be used by end-user programmers. We also summarize the results of our empirical investigations into the usefulness and effectiveness of these devices for promoting dependability in end-user programming.

WYSIWYT Testing

In our What You See Is What You Test (WYSIWYT) methodology, a user can test a spreadsheet incrementally as he or she develops it by simply validating any value as correct at any point in the process. Behind the scenes, these validations are used to measure the quality of testing in terms of a test-adequacy criterion. These measurements are then projected to the user via several different visual devices, to help them direct their testing activities.

For example, suppose a teacher is creating a student grades spreadsheet, as in Figure 1. During this process, whenever the teacher notices that a value in a cell is correct, she can check it off ("validate" it). The checkmark provides feedback, and later reminds the teacher that the cell's value has been validated under current inputs. (Empty boxes and question marks in boxes are also possible; both indicate that the cell's value has not been validated under the current inputs. In addition, the question mark indicates that validating the cell would increase testedness.)

A second, more important, result of the teacher's validation action is that the colors of the validated cell's borders become more blue, indicating that data dependencies between the validated cell and cells it references have been exercised in producing the validated values. From the border colors, the teacher is kept informed of which areas of the spreadsheet are tested and to what extent. Thus, in the figure, row 4's Letter cell's border is partially blue (purple), because some of the dependencies ending at that cell have now been tested. Testing results also flow upstream against dataflow to other cells whose formulas have been used in producing a validated value. In our example, all dependencies ending in row 4's Course cell have now been exercised, so that cell's border is now blue.

End-user software engineering employs a feedback loop supported by behind-the-scenes reasoning, with which the system and user collaborate to monitor dependability as the end user's program evolves.

If the teacher chooses, she can also view dependencies by displaying dataflow arrows between cells or between subexpressions in formulas. In Figure 1(b), she has chosen to view dependencies ending at row 4's Letter cell. These arrows follow the same color scheme as the cell borders.

A third visual device, a "percent tested" bar at the top of the spreadsheet, displays the percentage of dependencies that have been tested, providing the teacher with an overview of her testing progress.

Although the teacher need not realize it, the colors that result from placing checkmarks reflect the use of a *definition-use test adequacy criterion* [6] that tracks the data dependencies between cell formulas caused by references to other cells. Testing a program "perfectly" (well enough to guarantee detecting all faults) generally requires too many inputs; a test adequacy criterion provides a way to distribute a testing effort across elements of the program. In the spreadsheet paradigm, we say that a cell is fully tested if all its data dependencies have been covered by tests; those cells have their borders painted blue. Cells for which dependencies have not been fully covered have borders ranging from red to various shades of purple. The overall testing process is similar to the process used by professional programmers in "white box" unit testing, in which inputs are applied until some level of code coverage has been achieved. In the spreadsheet environment, however, the process is truly incremental, bearing some similarity to test-driven development approaches. These considerations, along with the testing theory underlying this methodology, are described in detail in [9].

Our teacher may eventually need to try different input values in certain cells in the spreadsheet, to cause other dependencies between formulas to come into play so their results can be checked. This process of conjuring up suitable inputs can be difficult, even for professional programmers, but help is available.

Help-Me-Test. To get help finding inputs to further test a cell, the teacher selects that cell and pushes the Help-Me-Test button HELP! in the spreadsheet toolbar. The system responds by attempting to generate inputs [5]. The system first constructs representations of the chains of dependencies that control the execution of particular data dependencies; then it iteratively explores portions of these chains, applying constrained linear searches over the spreadsheet's input space and data gathered through iterative executions. If the system succeeds, suitable input values appear in the cells, providing the teacher with new opportunities to validate. Our empirical results show that Help-Me-Test is typically highly effective and provides fast response [5].

Finding faults. Suppose in the process of testing, the teacher notices that row 5's Letter grade ("A") is incorrect. There must be some error in our teacher's formulas, but how shall she find it? This is a thorny problem even for professional programmers, and various technologies have been proposed to assist them. Some of these technologies build on information available to the system about successful and failed tests and about dependencies [11]. We are experimenting with approaches that draw from these roots [10]; here, we describe one of them.

Our teacher indicates that row 5's Letter grade is erroneous by placing an X mark in it. Row 5's Course average is obviously also erroneous, so she X's that one, too. As Figure 1(c) shows, both cells now contain pink interiors, but Course is darker than Letter because Course contributed to two incorrect values (its own and Letter's) whereas Letter contributed to only its own. These colorings reflect the likelihood that the cell formulas contain faults, with darker shades reflecting greater likelihood. The goal is to help the teacher prioritize which potentially suspicious formula to investigate first, in terms of their likelihood of contributing to a fault. Although this example is too small for the shadings to contribute a great deal, users in our empirical work who used the technique on larger examples did tend to follow the darkest cells. When they did so, they were automatically guided into dataflow debugging, which paid off in their debugging effectiveness.

Suppose that, with the help of the colorings, our teacher fixes the fault in the Course cell. (The weights in the weighted average did not add up to exactly 100%.) When she completes her edit the underlying algorithms partner with the spreadsheet evaluation engine in visiting affected cells in order to calculate the dependencies between formulas that might be affected by the changes. These dependencies are marked untested, and the rejuvenated screen display shows the resulting colors, directing the teacher's attention to portions of the spreadsheet that should now be retested.

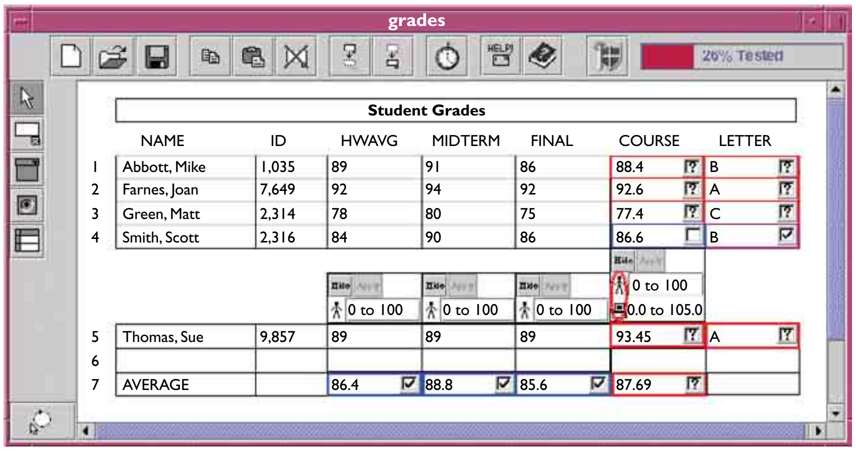
Assertions

Testing can reveal faults, but it may not reveal them all. Recent empirical work into human programming errors [7] categorized the types of errors participants made in introducing or attempting to remove faults. In that study, most errors were due to poor strategies and to attention problems such as paying attention to the wrong part of the program or working memory overload interfering with efforts to track down the fault. For professional programmers, assertions in the form of preconditions, postconditions, and invariants help with these issues, because these assertions can continuously attend to the entire program, reasoning about the properties the programmers expect of their program logic, and about interactions between different sections of the program. Our approach to assertions [3] attempts to provide these same advantages to end-user programmers such as the teacher.

These assertions are composed of Boolean expressions about cells' values. They look like enumerations of values and/or ranges of valid values, and these enumerations and ranges can also be composed ("and"ed and "or"ed together). For example, suppose the teacher had not noticed the fault in row 5's Course cell after all; we will show how assertions can be used to detect that fault. Suppose she creates assertions to continually monitor whether all numeric cells on row 5 will be between 0 and 100. To do so, she can either type ranges, as in Figure 2, or use a graphical syntax.

The assertions she enters (next to the stick figures) provide a cross-check that can automatically alert the teacher to even subtle faults such as getting the weights slightly wrong in the Course grade calculation. That power goes far beyond simply checking cell values against the user-entered assertions, and derives mainly from two sources: from aggressive participation by Help-Me-Test, and from propagation of some of the user-entered assertions to new system-generated assertions on downstream cells.

When assertions are present, Help-Me-Test's behavior is slightly different than we've described. For cells with only constants for formulas, it politely stays within the ranges specified by the assertions. But when cells with non-constant formulas have assertions, Help-Me-Test aggressively tries to derive input



Student Grades

	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4 ?	B ?
2	Farnes, Joan	7,649	92	94	92	92.6 ?	A ?
3	Green, Matt	2,314	78	80	75	77.4 ?	C ?
4	Smith, Scott	2,316	84	90	86	86.6 ☐	B ✓
						0 to 100	
			0 to 100	0 to 100	0 to 100	0.0 to 105.0	
5	Thomas, Sue	9,857	89	89	89	93.45 ?	A ?
6							
7	AVERAGE		86.4 ✓	88.8 ✓	85.6 ✓	87.69 ?	

Figure 2. When the teacher enters assertions, the system propagates them to deduce more assertions. In this case, a conflict was detected (circled in red), revealing a fault.

cell values that will violate those assertions on the downstream cells. Thus, the presence of assertions turns Help-Me-Test into an aggressive seeker of faults.

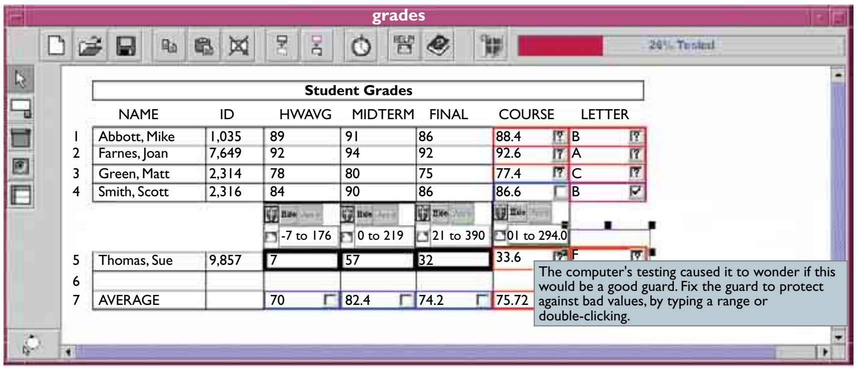
The propagation to system-generated assertions (for example, "0.0 to 105.0" next to the computer icon in Figure 2) produces three other ways assertions can semiautomatically identify faults. First, the system automatically monitors all values as they change, to see if they violate any of the assertions. Whenever a cell's value does violate an assertion, the system circles the value in red. For example, whenever the student's Course does not fall between 0 and 100, the system will circle it. Second, assertions might conflict with each other, as in Figure 2, in which case the system will circle the conflict in red. Conflicts indicate that either there is a fault in the cell's formula, or there are erroneous user-entered assertions. Third, the system-generated assertions might look wrong to the user, again indicating the presence of formula faults or user-entered assertion errors. All three ways to identify faults have been used successfully by end users. For example, in an empirical study [3], the participants using assertions were significantly more effective at debugging spreadsheet formulas than were participants without access to assertions.

The Surprise-Explain-Reward Strategy

A key to the power of assertions is the propagation aspect, which can happen only if there is an initial source of assertions from which to propagate. In some cases, initial sources of assertions might themselves be derivable (such as through statistical monitoring of input data [8] or based on nearby labels and annotations [4]). However, in other cases, the only possible source is the teacher herself. Still, it does not seem reasonable to expect the teacher to seek out an assertions feature in a spreadsheet environment. Given end users' unfamiliarity with quality control methods for software, strategies must be devised by which end-user software engineering approaches capture the interest of end-user programmers and motivate them to take appropriate steps that will enhance their software's correctness.

We have devised a strategy that aims to motivate end users to make use of software engineering devices, and to provide the just-in-time support needed to effectively follow up on this interest. Our strategy is termed Surprise-Explain-Reward [12]. It aims to choose timely moments to inform end users of the benefits, costs, and risks [1] of the software engineering devices available and of potential faults in the spreadsheet, so they can make informed choices about what actions to take next. It uses the element of surprise to attempt to arouse the curiosity of the end users, and if they become interested, the system follows up with explanations and, potentially, rewards.

For example, Help-Me-Test uses the element of surprise as a springboard in the Surprise-Explain-Reward strategy to introduce users to assertions. Whenever our teacher invokes Help-Me-Test, the system not only generates values for input cells, but also creates (usually blatantly incorrect, so as to surprise) "guessed" assertions to place on these cells. For example, in Figure 3, when the teacher selected row 5's Letter cell and pushed Help-Me-Test, while generating new values (indicated by thickened borders), Help-Me-Test also guessed some assertions. These guessed assertions, which we'll refer to as HMT assertions (because they are generated by Help-Me-Test), are intended to surprise the teacher into becoming curious about assertions. She can satisfy her curiosity using tool tips, as in Figure 3, which explain to her the meaning and rewards of assertions. If she follows up by accepting an HMT assertion (either as guessed or



Student Grades

	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4	B
2	Farnes, Joan	7,649	92	94	92	92.6	A
3	Green, Matt	2,314	78	80	75	77.4	C
4	Smith, Scott	2,316	84	90	86	86.6	B
			-7 to 176	0 to 219	21 to 390	01 to 294.0	
5	Thomas, Sue	9,857	7	57	32	33.6	F
6							
7	AVERAGE		70	82.4	74.2	75.72	

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.

Figure 3. While generating new values that will help increase testedness of row 5's Letter cell, Help-Me-Test also guessed some assertions.

	Number and types of studies	Populations studied	Main results
WYSIWYT testing	5 summative	End users, computer science students, spreadsheets	WYSIWYT was associated with more effective and efficient testing and debugging. End users with WYSIWYT tested more than those without WYSIWYT. WYSIWYT helped reduce overconfidence in spreadsheet correctness, but did not completely resolve this issue.
Help-Me-Test	2 formative, 2 summative	End users, spreadsheets	End users tended to test as much as they could without help initially, but when they eventually turned to Help-Me-Test, they commented favorably about it, and continued to use it. Users were willing to wait a long time for Help-Me-Test to try to find a value, and in the circumstances when it could not, they did not tend to lose confidence in the system. Users did not always make correct decisions about which values were right and which were wrong. Help-Me-Test algorithms were usually able to generate new test values quickly enough to maintain responsiveness.
Fault Localization	3 formative	End users	Different fault localization heuristics had very different advantages early in users' testing processes. Although some of the techniques tended to converge given a lot of tests, users did not tend to run enough tests to reach this point. When users made mistaken decisions about value correctness, their mistakes almost always assumed too much correctness (not too little correctness). Early computations, before the system has much information collected, may be the most important for shaping users' attitudes about the usefulness of fault localization devices. Those who used the technique tended to follow dataflow strategies about twice as much as the other participants, and the dataflow strategy was the only one tied to identification of "non-local" faults.
Assertions	2 formative, 1 summative	End users	End users using assertions were more effective and faster at debugging. Assertions were usable by end users.
Surprise-Explain-Reward	2 formative, 1 summative	End users	Comfort level and experience with the spreadsheet paradigm were important factors in determining whether "surprises" were motivating (interesting, arousing curiosity) or demotivating (perceived as too costly or risky). Surprise-Explain-Reward was effective in encouraging end users to use assertions, without forcing use of assertions before the users were ready. The type of communication used to communicate "surprises" may critically affect users' problem-solving strategies and productivity.

Table 1. Empirical work to date into end-user software engineering devices. (More details about the studies are in [3, 5, 9, 10, 12] and at www.engr.oregonstate.edu/~burnett/ITR2000/empirical.html).

after editing it), the resulting assertion will be propagated as seen earlier in Figure 2. As a result, the system may detect some problems; if so, red circles will appear as in Figure 2. If the red circles identify faults, the circles (and assertions) also serve as rewards.

It is important to note that, although our strategy rests on surprise, it does not attempt to rearrange the teacher's work priorities by requiring her to do anything about the surprises. No dialog boxes pop up and there are no modes. HMT assertions are a passive feedback system; they try to win user attention but do not require it. If the teacher chooses to follow up, she can mouse over the assertions to receive an explanation, which explicitly mentions the rewards for pursuing assertions. In a behavior study [12], 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 them effectively.

We have conducted more than a dozen empirical studies related to end-user software engineering research. Some of the results of these studies have been discussed here; all main results are summarized in Table 1. Some of our studies were conducted early in the development of our end-user software engineering devices, so as to influence their design at early stages; these are labeled "formative." Other studies evaluated the effectiveness devices at much later stages; these are labeled "summative."

Conclusion

Giving end-user programmers ways to easily create their own programs is important, but it is not enough. Like their counterparts in the world of professional software development, end-user programmers need support for other aspects of the software life cycle. However, because end users are different from professional programmers in background, motivation, and interest, the end-user community cannot be served by simply repackaging techniques and tools developed for professional software engineers. Directly supporting these users in software development activities beyond the programming stage—while at the same time taking their differences in background, motivation, and interests into account—is the essence of the end-user software engineering vision. As our empirical results show, an end-user programming environment that employs the approaches we describe here can significantly improve the ability of end-user programmers to safeguard the dependability of their software. C

References

1. Blackwell, A. First steps in programming: A rationale for attention investment models. In *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments* (Arlington, VA, Sept. 3–6, 2002), 2–10.
2. Boehm, B. Abts, C., Brown, A., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, J., and Steece, B. *Software Cost Estimation with COCOMO II.* Prentice Hall PTR, Upper Saddle River, NJ, 2000.
3. Burnett, M., Cook, C., Pendse, O., Rothermel, G., Summet, J., and Wallace, C. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the International Conference on Software Engineering* (Portland, OR, May 3–10, 2003), 93–103.
4. Burnett, M. and Erwig, M. Visually customizing inference rules about apples and oranges. In *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments* (Arlington, VA, Sept. 3–6, 2002), 140–148.
5. Fisher, M., Cao, M., Rothermel, G., Cook, C., and Burnett, M. Automated test generation for spreadsheets. In *Proceedings of the International Conference on Software Engineering* (Orlando FL, May 2002), 141–151.
6. Frankl, P., and Weyuker, E. An applicable family of data flow criteria. *IEEE Trans. Software Engineering 14*, 100 (Oct. 1988), 1483–1498.
7. Ko, A. and Myers, B. Development and evaluation of a model of programming errors. In *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments* (Auckland, NZ, Oct. 28–31), 7–14.
8. Raz, O., Koopman, P., and Shaw, M. Semantic anomaly detection in online data sources. In *Proceedings of the International Conference on Software Engineering* (Orlando, FL, May 19–25, 2002), 302–312.
9. Rothermel, G., Burnett, M., Li, L., DuPuis, C., and Sheretov, A. A methodology for testing spreadsheets. *ACM Trans. Software Engineering and Methodology 10,* 1 (Jan. 2001), 110–147.
10. Ruthruff, J., Creswick, E., Burnett, M., Cook, C., Prabhakararao, S., Fisher II, M., and Main, M. End-user software visualizations for fault localization. In *Proceedings of the ACM Symposium on Software Visualization* (San Diego, CA, June 11–13, 2003), 123–132.
11. Tip, F. A survey of program slicing techniques. *J. Programming Languages 3,* 3 (1995), 121–189.
12. Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M., and Rothermel, G. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems.* (Ft. Lauderdale, FL, Apr. 3–10, 2003), 305–312.

Margaret Burnett (burnett@eecs.orst.edu) is a professor in the School of Electrical Engineering and Computer Science at Oregon State University, Corvallis, OR.
Curtis Cook (cook@eecs.orst.edu) is Professor Emeritus in the School of Electrical Engineering and Computer Science at Oregon State University, Corvallis, OR.
Gregg Rothermel (grother@eecs.orst.edu) is an associate professor in the School of Electrical Engineering and Computer Science at Oregon State University, Corvallis, OR.

This work was supported in part by NSF under ITR-0082265 and in part by the EUSES Consortium via NSF's ITR-0325273. Collaborators on this project are listed at www.engr.oregonstate.edu/~burnett/ITR2000.