Test Reuse in the Spreadsheet Paradigm

Marc Fisher II, Dalai Jin, Gregg Rothermel, Margaret Burnett

Computer Science Department
Oregon State University
Corvallis, Oregon
grother@cs.orst.edu

Abstract

Spreadsheet languages are widely used by a variety of end users to perform many important tasks. Despite their perceived simplicity, spreadsheets often contain faults. Furthermore, users modify their spreadsheets frequently, which can render previously correct spreadsheets faulty. To address this problem, we previously introduced a visual approach by which users can systematically test their spreadsheets, see where new tests are required after changes, and request automated generation of potentially useful test inputs. To date, however, this approach has not taken advantage of previously developed test cases, which means that users of the approach cannot benefit, when re-testing following changes, from prior testing efforts. We have therefore been investigating ways to add support for test re-use into our spreadsheet testing methodology. In this paper we present a test re-use strategy for spreadsheets, and the algorithms that implement it, and describe their integration into our spreadsheet testing methodology. We report results of a case study examining the application of this strategy.

1 Introduction

Spreadsheet languages are widely used by a variety of end users to perform important tasks, such as tax calculations, budget management, and quality assessments of pharmaceutical products. The spreadsheets these end users create steer important decisions that may affect the welfare or safety of individuals, or even of large segments of society. The spreadsheet language paradigm is also a subject of ongoing research; for example, there is research into using spreadsheet languages for matrix manipulation problems [28], for scientific visualization [9] for providing steerable simulation environments for scientists [5], and for specifying full-featured GUIs [17, 27].

Users of spreadsheet languages "program" by specifying cell formulas. Each cell's value is defined by that cell's formula, and as soon as the user enters a formula, it is evaluated and the result is displayed. In essence, providing these spreadsheet formulas is an example of first-order functional programming.

It is important that spreadsheets function correctly, but research shows that they often contain faults. A survey of the literature [21] reports, for example, that in four field audits of operational spreadsheets, faults were found in 20.6% of the spreadsheets audited; in eleven experiments in which participants created spreadsheets, faults were found in 60.8% of those spreadsheets; in four experiments in which participants inspected spreadsheets for faults, an average of 55.8% of faults were missed. Research has also shown that spreadsheet users tend to have unwarranted confidence in the correctness of their spreadsheets [2, 30].

In spite of such evidence, until recently, little work had been done to help end users assess the correctness of their spreadsheets. Thus, we have been developing a testing methodology for spreadsheets termed the "What You See Is What You Test" (WYSIWYT) methodology [22, 24, 25]. The WYSIWYT methodology provides feedback about the "testedness" of cells in spreadsheets in a way that is incremental, responsive, and entirely visual. Empirical studies have shown that this methodology can help users test their spreadsheets more adequately and efficiently, and reduce end-user overconfidence [15, 26]. In subsequent work, we extended this approach to large grids of cells [6, 7], and to incorporate automated test input generation [12].

The incremental, responsive nature of the spreadsheet paradigm makes spreadsheets highly malleable, and spreadsheet users often make small changes to formulas in established spreadsheets. Such changes can render previously correct spreadsheets faulty, and merit re-testing. Our WYSIWYT methodology helps with this by indicating the areas of spreadsheets affected by changes; end users can retest those areas.

Such retesting, however, can involve considerable effort. This is due partly to the difficulty of finding test inputs that

exercise affected areas of spreadsheets, a task that automated test input generation can help with. However, automated test input generation may not produce outputs that can be easily validated (judged correct), forcing end users to repetitively call input generators until more suitable inputs are found, or even to generate inputs manually. When a spreadsheet is changed, it may be much simpler to re-use test cases that users have developed and found useful previously; for such test cases the correctness of outputs can be more easily established.

Further, research shows that spreadsheets are very often used and shared by other users, and passed on to still other users [18]. Sometimes these users must test a spreadsheet they have been given in order to verify its suitability for their purposes [18]. If they then decide to further adapt the spreadsheet to their own requirements, further re-testing is required. Asking users to test or re-test "from scratch" a spreadsheet they did not write wastes resources, and puts the onus of testing on persons who may have the greatest difficulties finding useful test inputs. In such a scenario, saved test cases embedded in the spreadsheet by an experienced spreadsheet developer might more easily be selectively re-used by end users following their modifications.

Finally, production spreadsheets are processed by commercial spreadsheet engines that are periodically re-released to provide new functionality or operate on new platforms. New releases of these engines, however, can cause spreadsheets to function differently than previously; thus, organizations that use spreadsheets for safety-critical tasks insist on revalidating their spreadsheets on new releases of spreadsheet engines, prior to allowing their use on those new releases.[1] Such revalidation would be greatly aided by the ability to re-use existing saved test suites.

To date, our WYSIWYT methodology has not taken advantage of previously developed test cases, but as the preceding scenarios suggest, there are many reasons for doing so. Further, although test re-use has been frequently addressed for the imperative paradigm, particularly in the context of regression testing, (e.g. [8, 16, 19, 23, 29]) it has not yet been considered for the spreadsheet language paradigm.

We have therefore been investigating ways to add support for test re-use into our spreadsheet testing methodology. We have developed algorithms for saving test cases in spreadsheet programs that operate in concert with the WYSIWYT methodology, algorithms for change impact analysis that determine test cases of interest following modifications, and strategies for enabling end users to re-use those test cases. We have implemented our test re-use strategies within the Forms/3 research spreadsheet environment. This paper presents our approach, describing differences between that approach and those explored previously in the imperative programming paradigm, and reports results of a case study examining factors relevant to the approach's cost-effectiveness.

[1] Personal communication, Scott Hutchinson, Amgen Corp..

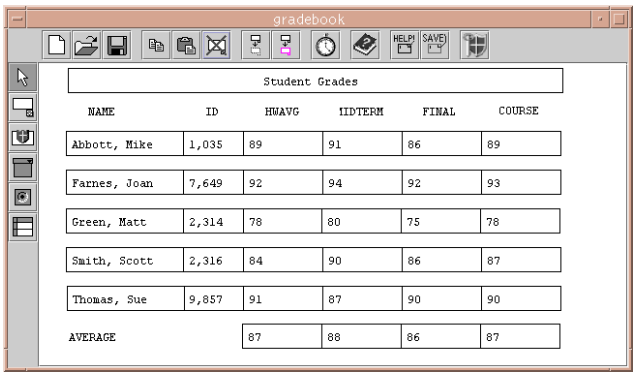

Figure 1. Spreadsheet for calculating grades.

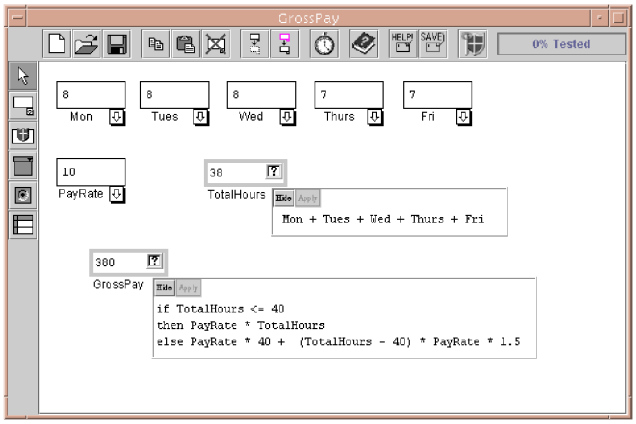

Figure 2. Forms/3 spreadsheet GrossPay.

2 The WYSIWYT Methodology

In this paper, we present examples of spreadsheets in the research language Forms/3 [3, 4]. Figure 1 shows a traditional-style spreadsheet used to calculate student grades in Forms/3. The spreadsheet lists several students, and several assignments performed by those students. The last row in the spreadsheet calculates average scores for each assignment, the rightmost column calculates weighted averages for each student, and the lower-right cell gives the overall course average (formulas not shown).

Figure 2 shows an example of a Forms/3 spreadsheet, `GrossPay`, which calculates an employee's weekly pay given a payrate and the hours they worked during that week. As the figure shows, in Forms/3, cells aren't restricted to grids, and the user can display or hide formulas as desired.

In our "What You See Is What You Test" (WYSIWYT), methodology for testing spreadsheets [22, 25, 26], as a user incrementally develops a spreadsheet, he or she can also test

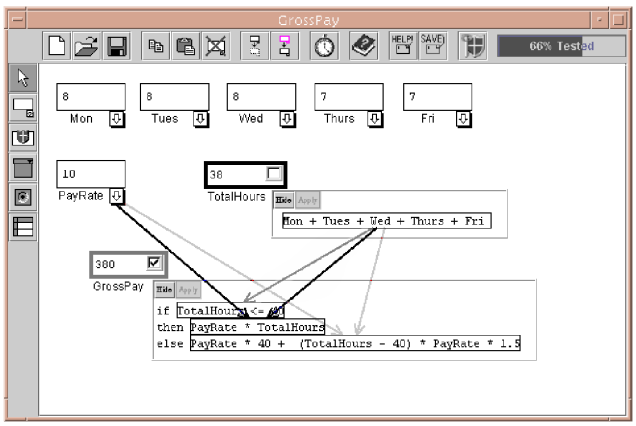

Figure 3. Spreadsheet GrossPay with testing information displayed after a user validation.

that spreadsheet incrementally. As the user changes cell formulas and values, the underlying engine automatically evaluates cells, and the user validates the results displayed in those cells. Behind the scenes these validations are used to measure the quality of testing in terms of a dataflow adequacy criterion, which tracks coverage of interactions between cells caused by cell references.

The following example illustrates the process from the user's perspective. It is important to understand the methodology's attributes from this perspective because they lead to requirements upon the test reuse strategy for spreadsheets that are not usual for traditional programming languages.

Suppose the user constructs the `GrossPay` spreadsheet by entering cells and formulas, reaching the state shown in Figure 2. Note that at this point, all cells other than input cells have red borders (light gray in this paper), indicating that their formulas have not been (in user terms) "tested". (Input cells are cells whose formulas contain no references and are, by definition, fully tested; thus, their borders are thin and black to indicate to the user that they aren't testable.)

Suppose the user looks at the values displayed on the screen and decides that cell `GrossPay` contains the correct value, given the current input values. To communicate this fact, the user clicks on the decision box in the upper right corner of that cell. One result of this action, shown in Figure 3, is the appearance of a checkmark in the decision box, indicating that the cell's output has been validated under the current inputs. (Two other decision box contents, empty and question mark, are possible. An empty decision box and a question mark each indicate that the cell's output has not been validated under the current inputs. In addition, the question mark indicates that validating the cell would increase testedness.)

A second result of the user's "validation" action is that the colors of the validated cell's borders become more blue (darker gray in this paper), indicating that interactions caused by references in that cell's formula have been exercised in producing validated outputs. In the example, in the formula for `GrossPay`, references in the `then` clause have now been thus exercised, but references in the `else` clause have not; thus, that cell's border is partially blue. Testing results also "propagate" to other cells whose formulas have been used in producing a validated value; in our example, all interactions ending at references in the formula for `TotalHours` have been exercised, hence, that cell's border is now fully blue (black in this paper).

If the user chooses, they can also view interactions caused by cell references by displaying dataflow arrows between cells or subexpressions in formulas; in the example, the user has chosen to view interactions ending at cell `GrossPay`. These arrows depict testedness information at a finer granularity, following the same color scheme as for the cell borders.

If the user next modifies a formula, interactions potentially affected by this modification are identified by the system, and information on those interactions is updated to indicate that they require retesting. The updated information is immediately reflected in changes in visual indicators (e.g., replacement of blue border colors by less blue colors).

Although a user of our methodology need not be aware of it, the methodology is based on the use of a dataflow test adequacy criterion adapted from the *output-influencing-all-du-pairs* dataflow adequacy criterion defined for imperative programs [10]; for brevity we call our adaptation of this criterion the *du-adequacy* criterion. We precisely define this criterion in [22]; here, we summarize that presentation to provide a basis for discussion of our test reuse strategy.

The du adequacy criterion is defined in terms of an abstract model of spreadsheets called a *cell relation graph* (CRG). Figure 4 shows the CRG for spreadsheet GrossPay. A CRG consists of a set of *cell formula graphs* (enclosed in rectangles in the figure) that summarize the control flow within formulas, connected by edges (dashed lines in the figure) summarizing data dependencies between cells. Each cell formula graph is a directed graph, similar to a control flow graph for imperative languages, in which each node represents an expression in a cell formula and each edge represents flow of control between expressions. There are three types of nodes in a cell formula graph: entry and exit nodes, representing initiation and termination of the evaluation of the formula; definition nodes, representing simple expressions that define the value of a cell; and predicate nodes, representing predicate expressions in cell formulas. Two edges extend from each predicate node: these represent the true and false branches of the predicate expression.

A *definition* of cell C is a node in C's formula graph representing an expression that defines C, and a *use* of C is either a *computational use* (a non-predicate node that

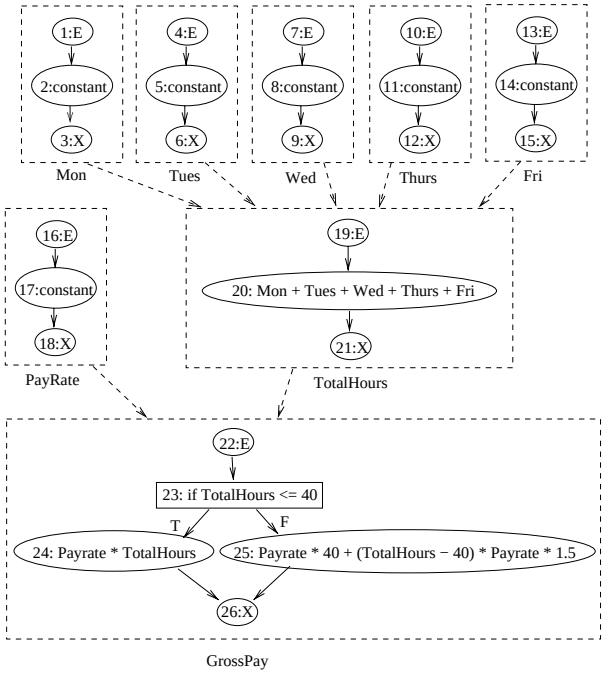

Figure 4. Cell relation graph for GrossPay

refers to C) or a *predicate use* (an out-edge from a predicate node that refers to C). A *definition-use association* (*du-association*) links a definition of C with a use of C which that definition can reach. A du-association is *exercised by a test* when inputs have been found that cause the expressions associated with its definition and its use to be executed, and where this execution produces a value in some cell that is pronounced "correct" by a user validation. Under the du-adequacy criterion, testing is adequate when each du-association in a spreadsheet has been exercised by at least one test case.[2]

In this model, a *test case* for a spreadsheet is a tuple (I,C), where C is a cell whose value the user has examined for correctness, and where I is a vector of input cells and their values, corresponding to those input cells in the spreadsheet on which C depends, directly or indirectly. Further, a *test* (the user's act of applying a test case) is an explicit decision by the user that C's value is correct or incorrect, given the current configuration I of input cells and values.

Because the process of manually identifying test inputs that will increase "blueness" is laborious, and may be difficult for users of spreadsheet languages, we have incorporated automated test case generation into our WYSIWYT methodology [12]. With our methodology, a user may select any combination of cells or arrows on the visible display, and then select the "Help Me Test" button in the Forms/3 toolbar.

At this point an underlying test case generation system responds, using a technique adapted from [11] to attempt to generate input cell values that will cause unvalidated du-associations in the area of interest to be exercised. If the system finds such inputs it applies them, and lets the evaluation engine update the display, indicating where the user can validate outputs and increase testedness. The user can then validate an output value, or can ignore the generated values (if, for example, they dislike the input set generated) and try again with the same or different cells or arrows selected, in which case the test case generation system attempts again using new seeded values. In our initial experiments using this methodology, the test generation system was able to find inputs to exercise nearly 99% of the feasible du-associations in a set of 1377 du-associations found in several spreadsheets [12].

3 Test Re-Use Methodology

In designing a test re-use methodology and incorporating it into the WYSIWYT approach, there are a number of requirements to consider.

First, to support the highly interactive spreadsheet programming environment and its reliance on immediate feedback, a test re-use methodology for spreadsheets must be considerate of users' time and attention. This means that it must be computationally efficient, and avoid unduly delaying users creating or modifying spreadsheets from making progress on the tasks they are trying to accomplish by using the spreadsheet in the first place. This also means that a methodology should not unduly control how users spend their attention [1]: attempting to force a user to perform a potentially lengthy regression testing task could be counterproductive to the user's spreadsheet efforts at that point in the spreadsheet's development. In such cases, users may abandon the use of the methodology.

It follows that a test reuse strategy for spreadsheets must be sufficiently precise in estimating affected areas of spreadsheets and making test cases available for re-use. If a user must perform actions that produce no apparent benefit, like retesting parts of a spreadsheet that do not apparently require retesting, they may become frustrated and choose not to re-test at all.

Further, although a test re-use strategy should be precise in the sense just described, it must also be conservative enough, in its estimates, to merit user trust. For example, in responding to user modifications, a re-use strategy should not miss areas of the spreadsheet that might be affected. Such an omission might cause end users to become (at least initially) overconfident in their spreadsheet, and later (on finding their confidence misplaced) to distrust the approach.

[2] It is not always possible to exercise all du-associations in a spreadsheet, and those that can not be exercised by any inputs are called *infeasible du-associations*. In general, the problem of identifying such du-associations is undecidable [13].

Finally, a test re-use methodology for spreadsheets should not, in general, require its users to practice (in any formal way) "software engineering", or have any formal knowledge of testing theory. The users of the methodology might not even think of "test cases" as "resources to be re-used", and they should not be expected to possess any recorded specifications that could be used to determine expected test results.

Keeping the foregoing considerations in mind, we have developed algorithms supporting test re-use in spreadsheet validation that are integrated with our WYSIWYT methodology. As we shall show in the following subsections however, the foregoing considerations cause our approach to differ from those that might be used in application to imperative programs and professional programmers.

Our first algorithm operates when an end user validates a cell, collecting information about the test case associated with that validation and storing it for later use. Our second algorithm operates when an end user changes a cell formula, determining impacted portions of the spreadsheet and effects on testedness, and judging whether existing test cases may be applicable to retesting the impacted portions. (These first two algorithms partially replace, and partially augment, previously presented WYSIWYT algorithms not supporting test re-use; portions of the overall approach not germane to an understanding of the test re-use issue that is our focus in this paper are omitted for simplicity.) The third algorithm operates when an end user requests help retesting the spreadsheet, replaying saved test cases so that the user can use them to revalidate the spreadsheet. The sections that follow present the data structures used by these algorithms, and then present each of the algorithms in turn.

3.1 Data Structures

Our test reuse algorithms rely on the data structures shown in Table 1 to record information on test cases and testedness and support test re-use; these data structures are maintained within the spreadsheet engine, and in most cases, updated incrementally as end users create, modify, and validate their spreadsheets. The data structures contain (1) information kept for each cell, (2) information kept for each test case, and (3) information kept for the spreadsheet. The data in these structures is also saved with a spreadsheet as it is filed out, and restored when it is read in.

Information kept per cell	
Cell.Name	identifier for *Cell*
.Value	a value displayed in *Cell*
.Trace	definition and use nodes exercised in *Cell*'s most recent execution
.ReachingTests	set of *Test*s that exercise *Cell*
Information kept per test case	
Test.InputCells	set of (*Cell.Name*,*Cell.Value*) pairs serving as inputs for *Test*
.ValidatedCell	(*Cell.Name*,*Cell.Value*) pair for the cell validated in *Test*
.ReachedCells	set of *Cell*s exercised by *Test*
.ValidatedDUs	set of du-associations *Test* validates
Information kept for the spreadsheet	
DUTable	hash table of validation counts, indexed by du-associations
ImpactedTests	set of *Test*s impacted by changes, and that might be rerunnable

Table 1. Data structures for test re-use.

3.2 Saving Test Case Information

When an end user performs a test by validating a cell in their spreadsheet, we walk backwards through the CRG for that spreadsheet, updating information associated with the test and its coverage, and saving the test case defined by the users' action. Our algorithm, `Validate` (Figure 5), takes as input a cell *C* validated by an end user. `Validate` first creates a new *Test* structure to represent the test case just created by this validation, initializing its fields to empty, and setting its *ValidatedCell* field to (*C.Name*,*C.Value*). The algorithm then proceeds in two phases, invoking procedures `GatherInputs` and `ValidateCoverage` in turn.[3] Each invocation is preceded by `InitWalk`, which initializes an integer flag used to mark cells "visited" and avoid revisiting cells during the ensuing walk.[4]

`GatherInputs`, called with validated cell *C* and test case *T*, performs a static backwards slice on cell dependence edges in the CRG, locating (recursively) all input cells whose values could affect *C*. (We refer to cells affecting *C* as *producers* of *C*. *Direct producers* of *C* — maintained by the spreadsheet engine and returned by a call to a function `DirectProducers` in line 11 — are producers explicitly listed in *C*'s formula.)[5]

`ValidateCoverage`, called with validated cell *C* and test case *T*, performs a dynamic backwards slice on du-associations in the CRG starting at *C*, recursively visiting each cell reached by *T*. Specifically, for each cell reached,

[3] These two phases, each performing a graph walk on the CRG, can be merged into a single graph walk with the same worst-case cost as that incurred by performing the phases separately, but with a lower constant; however, we present them as separate phases for simplicity.

[4] This integer flag is initialized to zero by the spreadsheet engine at startup, and incremented on each subsequent call to `InitWalk`; algorithms that walk the CRG use this flag to mark cells "visited" and avoid revisiting cells during a given walk. This eliminates the need to reset "visited" information on all cells prior to each walk, except in the unlikely event that the integer flag reaches its maximum value during a session using the spreadsheet engine.

[5] Using a *static* rather than a *dynamic* slice in this phase reduces the number of test cases that must be discarded as *obsolete* following a modification – the larger input cell sets recorded by the static slice reduce the likelihood that changes to a formula will increase the set of input cells on which that formula depends, making test cases associated with that cell non-applicable. We discuss test obsolescence and our handling of it in Section 3.4.

```
algorithm Validate(C)
input     C : cell
 1.  T = new Test
 2.  T.ValidatedCell = (C.Name,C.Value)
     /* Phase 1 */
 3.  InitWalk
 4.  GatherInputs(C,T)
     /* Phase 2 */
 5.  InitWalk
 6.  ValidateCoverage(C,T)

procedure GatherInputs(C,T)
input  C : cell; T : test case
 7.  MarkVisited(C)
 8.  if IsInput(C)
 9.    T.InputCells = T.InputCells ∪ {(C.Name, C.Value)}
10.  else
11.    for each cell D ∈ DirectProducers(C)
12.      if not Visited(D)
13.        GatherInputs(D,T)
14.      endif
15.    endfor
16.  endif

procedure ValidateCoverage(C,T)
input  C : cell; T : test case
17.  MarkVisited(C)
18.  C.ReachingTests = C.ReachingTests ∪ T
19.  T.ReachedCells = T.ReachedCells ∪ C
20.  for each use u in C.Trace
21.    D = the cell referenced in u
22.    d = the current definition of D ∈ D.Trace
23.    increment DUTable(d,u)
24.    T.ValidatedDUs = T.ValidatedDUs ∪ {(d,u)}
25.    if not Visited(D)
26.      ValidateCoverage(D,T)
27.    endif
28.  endfor
29.  UpdateDisplay(C)
```

Figure 5. Algorithm for collecting test information when the user validates a cell.

the procedure adds *T* to that cell's list of reaching tests (line 18), and adds that cell to *T*'s list of reached cells (line 19). The procedure next uses trace information maintained by WYSIWYT (collected and maintained by the spreadsheet engine during its operations to update the display following formula changes) to locate the du-associations ending at the cell reached that were exercised by *T*. For each such du-association (*d*,*u*), the procedure increments the coverage count for (*d*,*u*) (line 23), and adds (*d*,*u*) to the list of du-associations covered by *T* (line 24). Finally, the procedure calls `UpdateDisplay` to cause the colors of the cell's border, and of any arrows ending at that cell and currently being displayed, to be updated.

As an example of how the algorithm works, consider the `GrossPay` spreadsheet shown in Figures 2 and 3, and its CRG shown in Figure 4. When the user clicks on the decision box for cell `GrossPay`, `Validate` is called for that cell. The algorithm first creates a new *Test* `T`, and sets `T`.*ValidatedCell* to (`GrossPay`, 380). It then calls `GatherInputs` on `GrossPay` and `T`. Since `GrossPay` is not an input cell, `GatherInputs` takes the else branch from line 8, and then iterates over the set `DirectProducers(GrossPay)`, containing cells `PayRate` and `TotalHours`, recursively calling `GatherInputs` on each. `PayRate` is an input cell, so `T`.*InputCells* is set to {(`PayRate`, 10)}. Next, the recursive call `GatherInputs(TotalHours,T)` iterates over `DirectProducers(TotalHours)`, adding (`Mon`, 8), (`Tues`, 8), (`Wed`, 8), (`Thurs`, 7), (`Fri`, 7) to `T`.*InputCells*.

`Validate` next calls `ValidateCoverage` with `GrossPay` and `T`. `ValidateCoverage` adds `T` to `GrossPay`.*ReachingTests*, and adds `GrossPay` to `T`.*ReachedCells*. It then iterates through the use nodes, (23, `TotalHours`, T), (24, `PayRate`), and (24, `TotalHours`), found in `GrossPay`.*Trace*. On the first use, (23, `TotalHours`, T), the procedure finds that the currently active definition of `TotalHours` is node 17, and increments the testedness counter in *DUTable*(17, (23, `TotalHours`, T)). It adds (17, (23, `TotalHours`, T)) to `T`.*ValidatedDUs*, and recursively calls `ValidateCoverage(TotalHours, T)`, which updates `TotalHours`.*ReachingTests* and `T`.*ReachedCells*, and iterates through the five uses in `TotalHours`. When the iteration in `ValidateCoverage(GrossPay, T)` reaches (24, `PayRate`), (and subsequently (24, `TotalHours`)), a similar procedure is performed (except that `ValidateCoverage` is not recursively called on `TotalHours` as it was already marked "visited").

The time complexity for `Validate` depends on the cost of `GatherInputs` and `ValidateCoverage`. The cost of the initial call to `GatherInputs` depends on the number of times it is recursively called and the cost of each call. Since `GatherInputs` marks each cell when visited, and is not recursively called on visited cells, the upper bound on the number of calls to `GatherInputs` is p, where p is the number of producers of the validated cell. Each call to `GatherInputs` iterates through each of a cell's direct producers even if it does not make a recursive call, so this bounds the iterations in a call to `GatherInputs` at d, where d is the maximum number of direct producers of a cell in the spreadsheet. Each iteration involves a set union; however, the sets unioned are always necessarily disjoint. If these sets are implemented as lists and the union is implemented as an append, each iteration can be accomplished in constant time. Thus, `GatherInputs` runs in time O(dp).

Similarly, the cost of an initial call to `ValidateCoverage` is determined by the number of recursive calls that follow and the cost of each call. Since `ValidateCoverage` marks each cell when visited and is not recursively called on visited cells, the upper bound on the num-

Type of Change	*Type of Cell*		
	input	*intermediate*	*output*
Cell Deletion	(1) Not possible, or no action needed. An input cell cannot be deleted if it is currently being referenced; an input cell not being referenced can be deleted, but due to other operations can have no test cases associated with it.	(4) Not possible. A cell cannot be deleted if it is currently being referenced.	(7) Treat as formula change to blank, and use algorithm `ProcessMod`
Formula Change	(2) No action needed or use algorithm `ProcessMod`. A formula change involving insertion of a new constant is simply a test execution; insertion of a formula that references other cells is handled by `ProcessMod`.	(5) Use algorithm `ProcessMod`.	(8) Use algorithm `ProcessMod`.
Cell Insertion	(3) No action needed. An input cell being inserted cannot have any referencing cells prior to insertion; thus, its insertion changes no test information.	(6) Cell insertion involves inserting blank cells, then updating formulas for those cells; thus, this situation reduces to entry (3) followed by entry (2).	(9) Cell insertion involves inserting blank cells, then updating formulas for those cells; thus, this situation reduces to entry (3) followed by entry (2).

Table 2. Actions per type of change

ber of calls to `ValidateCoverage` is p where p is the number of producers of the initial validated cell. Each call to `ValidateCoverage` with cell C and test case T iterates through each use that was reached in C on T, even if it does not require a recursive call, so this places an upper bound on iterations of the loop at lines 20-28 in a single call at u, where u is the maximum number of uses in a formula. Assuming an efficient hash table implementation and given that the sets unioned are disjoint and can be implemented as lists with the union implemented as a list append, each iteration of the loop can be accomplished in constant time. Thus, `ValidateCoverage` runs in time O(up).

3.3 Responding to Changes

When a user modifies a spreadsheet, we must update all testing-related information, in the spreadsheet, that may be affected, and determine test cases that may be used to re-test affected portions of the spreadsheet. The actions required depend on the type of modification. Table 2 categorizes the modification types we need to consider, which depend on whether the user adds a cell, deletes a cell, or modifies an existing cell's formula, and on whether the cell in question is an input cell, intermediate cell, or output cell (cell not referenced by other cells).

For each entry in this table, we determined the actions necessary to handle the type of modification; the table summarizes those actions. (These actions hold for Forms/3 spreadsheets; a similar approach could be used to classify and determine how to handle modification types in other spreadsheet environments.) As the table and the following discussion show, several types of modifications do not require any handling, and those requiring handling are all processed by algorithm `ProcessMod`, shown in Figure 6. The actions and their use of the algorithm form a set of invariants which, together, ensure correct updating of testing-related information across possible change types, and ensure identification of test cases potentially useful for retesting.

Larger changes to a Forms/3 spreadsheet can only occur when users perform multiple smaller changes in succession; these can be handled by incrementally processing each of these smaller changes. Applying `ProcessMod` incrementally following each modification avoids problems that can occur when multiple modifications, which may interfere, are processed simultaneously [23].

`ProcessMod` relies on the fact that when a formula for a cell is edited, we already possess, attached to that cell, a list of all test cases that exercise that formula; this list includes all test cases that could be affected by the edit. The outer loop of the algorithm processes these test cases one at a time. For each test case *T*, the algorithm first removes *T* from the *ReachingTests* lists on each other cell it is associated with (line 3), and then adjusts *T*'s coverage information in *DUTable* (lines 6-8). `ProcessMod` saves the names of those cells (line 4) for use later in updating the display (line 13). The algorithm also sets *ValidatedDUs* and *ReachedCells* information for *T* to empty, since it cannot predict the behavior of the test case following modifications.

In line 11, `ProcessMod` lists *T* as impacted. Note that at this stage in our approach, such test cases are simply inserted into *ImpactedTests*, where they may join other test

```
algorithm ProcessMod(C)
input     C : cell
 1.  foreach test case T ∈ C.ReachingTests
 2.    foreach cell D ∈ T.ReachedCells
 3.      D.ReachingTests = D.ReachingTests − {T}
 4.      UpdateDisplaySet = UpdateDisplaySet ∪ {D}
 5.    endfor
 6.    foreach (d,u) ∈ T.ValidatedDUs
 7.      decrement DUTable(d,u)
 8.    endfor
 9.    T.ValidatedDUs = ∅
10.    T.ReachedCells = ∅
11.    ImpactedTests = ImpactedTests ∪ {T}
12.  endfor
13.  UpdateDisplay(UpdateDisplaySet)
```

Figure 6. Algorithm for selecting re-usable tests and updating testedness information.

cases placed there following previous modifications. Further processing of these test cases, however, is postponed to the test re-use phase, as described in Section 3.4. Note further that the test case's *InputCells* and *ValidatedCell* information are not altered by `ProcessMod`, even though, depending on the modification, that information may no longer be valid. This will also be explained in Section 3.4.

By identifying as impacted all test cases that exercise modified formulas, `ProcessMod` ensures selection of all test cases whose execution traces can differ as a result of the modification. Following results presented in [23], this allows the algorithm to not omit, in this stage of our methodology, test cases that could expose faults related to the user's formula modification. However, adjusting coverage data to reflect removal of such test cases also ensures that the algorithm will err only conservatively in estimating the impact of the modification on testedness, and in displaying affected areas of the spreadsheet to the user. As stated earlier, this is an important factor in limiting user distrust.

As an example, suppose an end user modifies the formula for `TotalHours` in spreadsheet `GrossPay`. This modification is handled by entry (5) in Table 2, resulting in a call to `ProcessMod(TotalHours)`. `ProcessMod` iterates over the set of test cases in `TotalHours`.*ReachingTests*. Suppose this set contains just the single test case described in Section 3.2. The algorithm decrements the coverage counts associated with this test case in *DUTable*, and removes it from the *ReachingTests* lists of all cells it is associated with. The test case is placed in *ImpactedTests*, and finally, the display is updated for all affected cells.

The time complexity for `ProcessMod` is determined by the number of iterations of each of its loops and the cost of each iteration. The outermost loop iterates t times where t is the number of test cases in $C.ReachingTests$. Within this loop there are two separate nested loops. The first (lines 2-5) iterates at most n times where n is the maximum number of cells in the $ReachedCells$ field of a test case. The second loop (lines 6-8) iterates for each du-association in the $ValidatedDUs$ field of a test case. Each test case can validate at most O(nu) du-associations, where u is the maximum number of uses for the cells reached by a test case. An efficient hash table implementation holds the cost of each such iteration to O(1). Combining the above costs yields a total cost of O($tn*$ max(u, cost of set operations)).

3.4 Re-using Test Cases

Our algorithms for saving test cases and for determining impacted test cases following modifications provide two classes of test cases that can be used to re-validate spreadsheets, in two different scenarios.

1. Test cases that have been inserted into *ImpactedTests* are related to modifications, and may be useful in revalidating du-associations associated with changes.
2. All saved test cases can be replayed, allowing a spreadsheet to be re-validated in full when ported to a new spreadsheet environment.

We describe these two test classes and scenarios in turn.

3.4.1 Retesting Following Changes

Our algorithms cause test cases that might be useful in retesting modified formulas and re-establishing coverage to be placed in *ImpactedTests*. Replaying such test cases involves applying the saved input values to appropriate input cells, and allowing the user to re-validate output cells. There are several aspects of this process to consider.

Test case replay strategy. The first aspect involves choosing a test case replay strategy and designing an interface that implements it. One strategy, based on typical techniques used in the imperative programming paradigm [19], is to require the user to replay and revalidate all impacted test cases after each modification (or possibly after indicating that a set of related modifications has been completed). This strategy offers the greatest level of assurance and conservatism, by requiring the user to rerun all test cases that might be affected by their modifications, and might thus conceivably expose faults related to those modifications.

Reflecting on the considerations with which we began this section (Section 3), however, this strategy has drawbacks. Re-running all test cases identified by `ProcessMod` might require more time than a user is willing to spend. Further, it is possible that, due to changes in the spreadsheet or the order in which test cases are rerun, some test cases identified as "impacted" will not lead to increases in coverage, and this violates the restriction on forcing users to perform actions that produce no apparent benefit.

A second strategy involves requiring the user to revalidate all impacted test cases that could increase coverage.

These test cases can be determined automatically by the system by having it save current input values, apply the test inputs, activate the evaluation engine, and determine whether uncovered du-associations are covered, and only then update the display and allow the user to proceed with validation. (Considering existing imperative language approaches, this strategy is most similar to those presented in [14, 20].) While this approach solves the problem of perceived wasted effort, it violates the principle of allowing users to control how they spend their own attention [1].

The key drawback behind these two strategies is that the system does not really have the power to "require" testing-related actions, because an end user could simply turn off the feature or turn to a different spreadsheet environment that does not have such a requirement. The action must be deemed worthwhile to keep the user from doing this, and requirements that the user consider a large number of test case applications, or consider test case applications that are unproductive, is not likely to be deemed worthwhile.

A third strategy involving making only potentially useful test cases available without forcing any particular user action can be achieved by integrating test re-use with automated test case generation. This approach is triggered when the user requests help generating test inputs, which they might do following a modification to restore testedness in affected areas of the spreadsheet. Given such a request, rather than attempting to generate new inputs, the test case generation functionality first tries to find a rerunnable test case that will increase coverage of the portion of the spreadsheet indicated by the user. If a useful test case is found, the test generation facility applies that test case's inputs, displays the resulting changes, and the user can re-validate cells under that input set.

This third strategy avoids the drawbacks of the first two, and also avoids requiring a user to have any concept of a test case as an existing, re-usable resource, instead, integrating test re-use with the "Help-Me-Test" facility. We therefore implemented this third strategy.

Obsolete test cases. The second aspect of test case re-use to consider involves test cases that can no longer be applied as first recorded, due to changes in the spreadsheet. If the output cell associated with a test case is deleted, that test case can no longer be applied. If formula changes cause a cell to depend on new input cells, test cases formerly associated with that cell no longer come with all necessary inputs. In the literature on regression testing of imperative programs, such test cases are known as *obsolete* [16]. Such test cases could be detected and eliminated by `Process-Mod` when processing a modification; however, this adds to the expense of that algorithm, and since subsequent modifications may render a test case previously judged obsolete valid again, we defer the handling of these test cases to the test replay stage.

In our approach, when a user selects Help-Me-Test, as the test case generation facility iterates over *ImpactedTests* seeking useful test cases, it begins by determining whether each test case T it encounters is obsolete. To do this, the test case generation facility first determines whether the validated cell V associated with T still exists in the spreadsheet. If not, then T is obsolete and is discarded. If V exists, the generator determines the set of input cells that could affect V, using an algorithm similar to `GatherInputs`. If this set includes input cells not found in set *V.InputCells* associated with the test case originally, we do not know what values to use for those cells and cannot re-apply the test case transparently, so we consider it obsolete.[6] Obsolete test cases can be discarded from the data structures, or retained in case further modifications render them non-obsolete; to avoid a potentially excessive build-up of such test cases we take the first course of action.[7]

Output validation strategy. When test cases are saved, their validated output cell values are also saved; these values could be re-used in re-testing the spreadsheet after modifications. Such an approach is typical in testing imperative programs [19].

It is tempting to consider automatically checking test case outputs against saved outputs when they are re-run, and when saved outputs match new outputs, consider test cases to have automatically passed. This strategy, however, is inappropriate. An end user might modify a spreadsheet in order to change its functionality, such that previous inputs are expected to produce different outputs than previously. Thus, for some test cases, the fact that saved outputs match new outputs does not signal correctness, but rather, incorrectness. In general, in the absence of specifications, a test replay strategy cannot automatically infer that matching outputs signal either correctness or incorrectness.

An alternative strategy, when replaying previous test cases, is to display the previous test case output alongside the new test case output, to help the user validate the new output. It is not clear, however, whether such an approach would actually assist the user, or whether users would be inclined to make the potentially erroneous assumption that old values should match new, and that matching indicates correctness — an assumption that, as just discussed, may be erroneous.

[6] Alternative approaches are possible. If the input cell set has increased, we can apply those inputs that we know, leaving other inputs at their current values; and if the validated cell is now absent, the test case's input set might nevertheless still be applied. In one other scenario – the scenario in which the set of input cells now associated with a test case is a subset of those previously associated with it, we do choose such an alternative, and apply previous inputs to those input cells that still exist. Future work will examine the cost-benefits of these alternatives.

[7] As foreshadowed in Section 3.2, it is to reduce the incidence of obsolete test cases that the first phase of `Validate` uses a static rather than a dynamic slice to determine the input cell set associated with a test case.

For these reasons, we do not yet use previous test case outputs during test replay; we will first explore the ramifications of approaches empirically with end users before selecting an output re-use strategy.

3.4.2 Re-applying Tests

By saving all test case information with a spreadsheet when we save it to disk, and re-loading it when that spreadsheet is re-loaded, we also support the complete re-execution of saved test suites. End-users wishing to re-validate a spreadsheet for a new release of an evaluation engine can take advantage of this. This test case re-use scenario differs from the previous scenario in that it does imply that users be aware of test cases as objects, and consider saved test cases a resource; however, we expect that in certain contexts, for certain users, this will not be appropriate.

Also, in this context, re-use of output values may make more sense, and a utility that iterates through all saved test cases may be palatable to end users. Our future studies will consider this possibility.

4 Case Study

A primary goal of our test case re-use methodology is to help users re-test their spreadsheets following modifications. Determining whether the methodology achieves this goal will require empirical studies of users; however, before undertaking such studies we must first address a more fundamental question: namely, whether the methodology can in fact usefully select test cases for re-use. If the answer to this question is negative, there is no reason to pursue more expensive studies involving human subjects.

To assemble some initial data toward this question, we prototyped our test re-use methodology in Forms/3, in conjunction with our WYSIWYT and automated test case generation methodologies. We used this prototype to perform a case study of the application of our methodology to a large spreadsheet undergoing correction for errors.

4.1 Object of Study

As an object of study we selected a large spreadsheet, MBTI, that had been used in earlier empirical studies of automatic test case generation [12]. MBTI implements a version of the Myers-Briggs Type Indicator (a personality test), which tallies the scores and reports personality types based on integer answers to twenty questions. The spreadsheet contains 48 cells, 784 du-associations (780 feasible), 248 expressions, and 100 predicates.

We asked an experienced spreadsheet user unfamiliar with our test case reuse methodology and the anticipated case study to generate 14 different faulty versions of MBTI by inserting minor faults in the spreadsheet.

We used our test case generation system to generate a test suite for each faulty version through repeated applications of the generation algorithm, continuing until several applications failed to find new useful inputs. The resulting test suites, although not 100% adequate in covering all feasible du-associations in the erroneous spreadsheets, were nearly so, with coverage ranging from 93.11% to 98.85% of the du-associations in the spreadsheets. (The differences in testedness across spreadsheets can be attributed primarily to differences in feasible du-associations among faulty versions, and to differences in the ability of the test case generation system to cover the du-associations that differed across those faulty versions.)

4.2 Study Design

To investigate our research question we collected data on several measures related to our test re-use methodology and the potential usefulness of the test cases our algorithms identify both following modifications, and at the time test cases are re-run. These measures include:

- the number and percentage of test cases identified by our test re-use strategy of Section 3.4 as rerunnable following a modification to, and an end user's request to re-validate, a spreadsheet;
- the level of testedness that can be regained on the spreadsheet by rerunning all rerunnable test cases.

To evaluate our test reuse methodology relative to these measures, we wished to simulate a user performing a correction of a faulty formula in `MBTI`, and then re-running all re-runnable test cases that can add coverage. To do this, for each of our faulty versions of `MBTI`, we loaded that version, with all of its saved test cases, into the Forms/3 environment. We then corrected the fault in the spreadsheet, causing `ProcessMod` to be invoked and identify impacted test cases. Finally, we ran a script that automatically performed a "Help-Me-Test" operation repeatedly, invoking the test case generation technique until all test cases had been considered, and judged obsolete or, if able to add coverage, re-applied. Following each application of a re-used test case, we automatically validated the output cell associated with that test case. We repeated this process on each of the fourteen faulty versions of MBTI, in turn.

4.3 Evidence Analysis

Our case study involves only a single spreadsheet, and a set of modifications made to correct relatively small faults, and thus, cannot generalize to the range of situations in which we would like to apply our approach. Keeping this limitation in mind, however, the data gathered do let us draw some observations about the feasibility of the approach and its possible benefits.

1	2	3	4	5	6	7	8	9
Spreadsheet Version	*Total Test Cases*	*Initial Testedness*	*Testedness After Modification*	*Impacted Test Cases*	*Rerunnable Test Cases*		*Obsolete Test Cases*	*Final Testedness*
					Count	*Percentage*		
1	464	98.21	88.64	51	42	9.05	0	98.09
2	474	96.30	87.88	46	46	9.71	0	95.54
3	479	98.09	89.29	57	57	11.90	0	98.09
4	488	97.83	95.41	21	21	4.30	0	97.83
5	428	98.41	90.69	21	16	3.74	0	95.28
6	445	97.83	89.29	41	0	0.00	41	89.29
7	493	98.60	89.16	53	0	0.00	53	89.16
8	433	97.99	87.25	39	0	0.00	39	87.25
9	494	97.45	89.92	45	45	9.11	0	95.41
10	447	97.58	85.46	54	54	12.08	0	93.11
11	472	98.85	86.48	65	58	12.29	0	98.34
12	448	98.45	89.67	44	39	8.71	0	97.58
13	465	98.60	87.50	60	37	7.96	0	98.60
14	402	93.11	86.35	36	31	7.71	0	93.88

Table 3. Test re-use data gathered in case study on `MBTI`.

Table 3 presents the data collected in our study. For each faulty version of `MBTI`, the table lists the total number of associated test cases (column 2), the initial testedness these test cases achieved on the faulty version prior to correction of the fault (column 3), and the spreadsheet's reduced level of testedness following the fault correction (column 4). Column 5 shows the number of impacted test cases identified by `ProcessMod`. Columns 6 and 7 show the number and percentage (over all test cases) of test cases identified as rerunnable by the methodology, and column 8 shows the number of test cases identified as obsolete. Column 9 displays the final testedness achieved on the spreadsheet after re-running the rerunnable test cases.

First we consider the number of rerunnable test cases. There are two questions to consider here: are useful rerunnable test cases available, and is the number of such test cases a reduction over the entire test suite? For three of the 14 modified versions of `MBTI`, all impacted test cases were identified as obsolete. On the other 11 versions, however, rerunnable test cases were found, and their numbers represented only 3.74% to 12.29% of the entire test suite. Such results suggest that, at least for the types of corrective modifications considered, our test re-use approach can identify a reasonable number of test cases for re-use.

Columns 3, 4, and 9 let us consider the effectiveness of re-used test cases in re-establishing testedness after modifications. In six of the 14 cases considered, test re-execution restored testedness to within 1% of its original level (in three cases final testedness equaled, and in one case it exceeded, initial testedness). In all other cases except those where test cases were obsolete, final testedness fell within 5% of original testedness. This shows that when our test reuse methodology finds rerunnable test cases, those test cases can be effective at restoring testedness in portions of the spreadsheet affected by modifications, allowing end-users to re-achieve much of their initial coverage without creating new test cases.

5 Conclusions

We have presented a test re-use methodology for use with spreadsheets, and algorithms that implement it. The algorithms operate in concert with our WYSIWYT spreadsheet testing methodology, and are tightly integrated with the highly interactive spreadsheet programming environment, presenting testing-related information visually, and allowing end-users to re-use testing information without requiring formal understanding of testing. We have prototyped our methodology, and our case study shows that it can usefully select test cases for reuse.

Given these results, our next step in this research is the design and performance of additional experiments, including (1) experiments with a wider range of spreadsheets, including representatives of commercial spreadsheet applications, and (2) experiments involving human users of our methodology. Such studies will help us further investigate open questions about design strategies (such as re-use of output information), and ultimately, assess whether our methodology can be used effectively by end users on production spreadsheets.

Acknowledgements

We thank the Visual Programming Research Group and Curt Cook for their work on Forms/3 and on the WYSIWYT and Help-Me-Test methodologies, and for their feedback on our test re-use methodologies. This work has been supported by the National Science Foundation by ESS Award CCR-9806821 and ITR Award CCR-0082265 to Oregon State University.

References

[1] A. Blackwell. See what you need: Helping end users to build abstractions. *J. Vis. Lang. Computing*, 12(5):475–499, Oct. 2001.

[2] P. Brown and J. Gould. Experimental study of people creating spreadsheets. *ACM Trans. Office Info. Sys.*, 5(3):258–272, July 1987.

[3] 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. Funct'l. Prog.*, 11(2):155–206, 2001.

[4] M. Burnett and H. Gottfried. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Trans. Computer-Human Int.*, pages 1–33, Mar. 1998.

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

[6] 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. Softw. Eng.*, (to appear).

[7] M. Burnett, A. Sheretov, and G. Rothermel. Scaling up a "What You See is What You Test" Methodology to Spreadsheet Grids. In *Proc. 1999 IEEE Symp. Vis. Lang.*, pages 30–37, Sept. 1999.

[8] V. Chen, D. S. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proc. 16th Int'l. Conf. Softw. Eng.*, pages 211–220, May 1994.

[9] E. H. Chi, P. Barry, J. Riedl, and J. Konstan. A spreadsheet approach to information visualization. In *IEEE Symp. Info. Vis.*, Oct. 1997.

[10] E. Duesterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Proc. 2nd Irvine Softw. Symp.*, Mar. 1992.

[11] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Meth.*, 5(1):63–86, Jan. 1996.

[12] M. Fisher, M. Cao, G. Rothermel, C. R. Cook, and M. Burnett. Automated Test Case Generation for Spreadsheets. In *Proc. 24th Int'l. Conf. Softw. Eng.*, May 2002 (to appear, preprint available at http://www.cs.orst.edu/~grother).

[13] P. Frankl and E. Weyuker. An applicable family of data flow criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, Oct. 1988.

[14] M. J. Harrold and M. L. Soffa. An incremental approach to unit testing during maintenance. In *Proc. Conf. Softw. Maint.*, pages 362–367, Oct. 1988.

[15] V. B. Krishna, C. R. Cook, D. Keller, J. Cantrell, C. Wallace, M. M. Burnett, and G. Rothermel. Incorporating incremental validation and impact analysis into spreadsheet maintenance: An empirical study. In *Proc. Int'l. Conf. Softw. Maint.*, Nov. 2001.

[16] H. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, Oct. 1989.

[17] B. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *ACM CHI '91*, pages 243–249, Apr. 1991.

[18] B. Nardi. *A small matter of programming: Perspectives on end user computing*. The MIT Press, Cambridge, MA, 1993.

[19] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Comm. ACM*, 41(5):81–86, May 1998.

[20] T. Ostrand and E. Weyuker. Using dataflow analysis for regression testing. In *Sixth Annual Pacific Northwest Softw. Qual. Conf.*, pages 233–247, Sept. 1988.

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

[22] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Trans. Softw. Eng.*, pages 110–147, Jan. 2001.

[23] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), Aug. 1996.

[24] G. Rothermel, L. Li, and M. Burnett. Testing strategies for form-based visual programs. In *Proc. 8th Int'l. Symp. Softw. Rel. Eng.*, pages 96–107, Nov. 1997.

[25] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proc. 20th Int'l. Conf. Softw. Eng.*, pages 198–207, Apr. 1998.

[26] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[27] T. Smedley, P. Cox, and S. Byrne. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *Advanced Visual Interfaces '96*, May 1996.

[28] G. Viehstaedt and A. Ambler. Visual representation and manipulation of matrices. *J. Vis. Lang. Computing*, 3(3):273–298, Sept. 1992.

[29] F. I. Vokolos and P. G. Frankl. Pythia: A regression test selection tool based on textual differencing. In *Proc. 3rd Int'l. Conf. Rel., Quality, and Safety of Softw.-Intensive Sys.*, May 1997.

[30] E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, and C. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *ACM CHI'97*, pages 22–27, Mar. 1997.