Testing Homogeneous Spreadsheet Grids with the "What You See Is What You Test" Methodology[*]

Margaret Burnett, Andrei Sheretov, Bing Ren, and Gregg Rothermel
Department of Computer Science
Oregon State University, Corvallis, Oregon 97331
{burnett, andrei, ren, grother}@cs.orst.edu

Abstract

Although there has been recent research into ways to design environments that enable end users to create their own programs, little attention has been given to helping these end users systematically test their programs. To help address this need in spreadsheet systems—the most widely used type of end-user programming language—we previously introduced a visual approach to systematically testing individual cells in spreadsheet systems. However, the previous approach did not scale well in the presence of largely homogeneous grids, which introduce problems somewhat analogous to the array-testing problems of imperative programs. In this paper, we present two approaches to spreadsheet testing that explicitly support such grids. We present the algorithms, time complexities, and performance data comparing the two approaches. This is part of our continuing work to bring to end users at least some of the benefits of formalized notions of testing, without requiring knowledge of testing beyond a naive level.

Index terms: *software testing, spreadsheets, visual programming*

1. Introduction

There has been extensive research into effective testing in traditional programming languages in the imperative paradigm. However, there are few reports in the literature on testing in other paradigms, and no research (with the exception of our own previous work) that we have been able to locate on testing in spreadsheet systems. The spreadsheet paradigm includes not only

[*] An early version of portions of this paper previously appeared in conference form [6].

commercial spreadsheet systems, but also a number of research languages that extend the paradigm with features such as gestural formula specification [3, 17], graphical types [3, 32], visual matrix manipulation [31], high-quality visualizations of complex data [7], and GUI specification [19]. In this paper, we use the term *spreadsheet languages* to describe all such systems following the spreadsheet paradigm.

Despite the perceived simplicity of the spreadsheet paradigm, research shows that many spreadsheets contain faults. Field audits of real-world spreadsheets report faults in 20% to 90% of the spreadsheets audited, and these rates are consistent with spreadsheet model-building experiments in controlled lab settings [2, 9, 21, 30]. Even though spreadsheets often contain faults, few companies have policies or standards for developing, documenting, or testing spreadsheets. Similarly, few companies have procedures for verifying the correctness of spreadsheets [30]. Furthermore, users tend to be overconfident that their spreadsheets contain no faults. For example, in Brown and Gould's experiment [2], subjects were "quite confident" that the spreadsheets they created were accurate; yet analysis showed that 44% of the spreadsheets contained faults and that every subject made at least one error. These error rates are consistent with Galletta et al.'s, in which even expert users could find only about 50% of the faults placed in their spreadsheets [13].

There have been attempts to reduce the number of faults in spreadsheets. Many of these attempts are proposals to apply common software development practices. For example, Ronen et al. [24] propose a structured approach to designing spreadsheets. Their approach includes a proposed layout of the spreadsheet model and a spreadsheet flow diagram similar to the data flow diagrams in structured analysis that encourage structured top-down design. Another approach is for the spreadsheet to include tools that aid comprehension. For example, some versions of Microsoft Excel auditing tools have included cell precedent and dependent arrows [18]. Precedent arrows for a cell A point to A from all cells that are referenced in A's formula. Dependent arrows for a cell B point from B to all cells that reference B in their formulas. These approaches add information and structure which may help spreadsheet programmers avoid faults, but none of them support testing.

To address this problem, in previous work [25, 26, 27], we presented a testing methodology for spreadsheets termed the "What You See Is What You Test" (WYSIWYT) methodology. The WYSIWYT methodology provides feedback about the "testedness" of cells in simple spreadsheets in a manner that is incremental, responsive, and entirely visual. However, scalability to large grids (large, two-dimensional matrices of cells) was not addressed in that

grades

Student Grades

	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE
1	Abbott, Mike	1035	89	91	86	89 ✓
2	Farnes, Joan	7649	92	94	92	93 ?
3	Green, Matt	2314	78	80	75	78
4	Smith, Scott	2316	84	90	86	87
5	Thomas, Sue	9857	91	87	90	90
	AVERAGE		87 ?	88	86 ✓	87

CELL MATRIX RADIO OPTION Hide Form Help Cut Cell Paste Hide Test Data

Figure 1. Forms/3 `grades` spreadsheet. The user validated four of the cells, and then, to test further, entered a new input for Farnes's `HWAVG`. This new input changed the affected `Average` and `COURSE` cells' √s to ?s, because these cells no longer contain the values the user validated. The `COURSE` formulas (not shown) have an `if`-expression; since only one branch of it has been tested, the borders for the two `COURSE` cells that have been validated to date (those in the top two rows) are between red and blue (light gray and black, for color-blind users).

previous work. In this paper, we describe improvements that allow the WYSIWYT methodology to support testing of large grids of cells with shared or copied formulas.

We have integrated a prototype implementation of the WYSIWYT methodology into the research spreadsheet language Forms/3 [3, 4, 5], and the examples in this paper are presented in that language. In our prototype, every cell in the spreadsheet is considered to be untested when it is first created, except *input cells* (cells whose formulas may contain constants and operators, but no cell references and no `if`-expressions), which are considered trivially tested. For the non-input cells, "testedness" is reflected via border colors on a continuum from untested (red, or light gray in this paper) to tested (blue, or black in this paper).

The process is as follows. During the user's spreadsheet development, whenever the user notices a correct value, he or she lets the system know of this decision by *validating* the correct cell (clicking in the validation checkbox in its right corner), which causes a checkmark to appear, as in Figure 1. This communication lets the system track judgments of correctness, propagate the implications of these judgments to cells that contributed to the computation of the validated cell's value, and reflect this increase in "testedness" by coloring borders of the checked cell and its contributing cells more blue (darker gray to black). On the other hand, whenever the user notices an incorrect value, rather than checking it off, he or she eventually finds the faulty formula and fixes it. This formula edit means that affected cells will now have to be re-tested; the system is aware of which ones those are, and re-colors their borders more red (lighter gray), denoting more "untested".

As with programs in other languages, most spreadsheets can have an infinite number of inputs; hence, not all possible inputs can be tested, and a means must be provided for determining whether testing has been adequate. In our previous work, we developed an abstract model for simple spreadsheets with conventional expression-based formulas, and used it to define several test adequacy criteria [25, 26]. The strongest criterion we defined, *du-adequacy,* is the criterion we use in this paper to define when a spreadsheet has been tested "enough". We describe the model in Section 2 and extend the model and du-adequacy criterion as they relate to spreadsheet grids in subsequent sections. The border colors described above indicate the extent to which the du-adequacy criterion has been satisfied.

Thus, if the user manages to turn all the red (light gray) borders blue (black), the du-adequacy criterion has been satisfied. In our empirical work on simple spreadsheet cells, subjects were significantly more likely to achieve du-adequate coverage and do so efficiently using the WYSIWYT methodology than those not using it [28], du-adequate test suites were frequently significantly more effective at fault detection than random test suites [25], and subjects were significantly more likely to correctly eliminate faults using the WYSIWYT methodology than those not using it [8].

The methodology for testing spreadsheets as described above worked at the granularity of individual cells. However, most large grids in spreadsheets are fairly homogeneous, i.e., they consist of many cells whose formulas are identical except for some of the row/column indices. For example, suppose the spreadsheet in Figure 1 were expanded to calculate student grades for a class containing 500 students. There are two problems with applying the previous testing methodology to this kind of grid:

Problem 1: For the user, the problem is that each of the 500 course grade cells would have to be explicitly validated for all the borders to appear blue (black), denoting completely tested. The user is not likely to go to this much trouble for essentially identical cells; this would mean that the user would be burdened with keeping track of which cells "really" need testing and which (due to their similarities to other cells) do not.

Problem 2: For the system, the problem is that the performance of the testing subsystem depends on the number of cells. Hence, responsiveness is impaired by the presence of large grids.

For both the user and the system, these burdens seem inappropriate, given that the `Grades` spreadsheet's formulas with 500 students are exactly the same as those of the `Grades` spreadsheet with only 5 students. To address these problems, the previous methodology needed to be extended to explicitly support homogeneous grids. In addition, we imposed a "do no harm"

constraint, requiring that any such extensions not add significant overhead to testing spreadsheets that do not feature such large grids.

2. Background

2.1 Homogeneity of grids

A *grid* is a two-dimensional matrix of cells. Most commercial spreadsheet systems are entirely grid-based. The grids of particular interest in this work are largely homogeneous—i.e., most of their cells have identical formulas except perhaps for row/column indices. Thus, in this paper, the term *grid* implies some homogeneity, and the term *region* refers to a subgrid in which every cell has the same formula, except that row/column indices may differ.

A spreadsheet language requires knowledge of the homogeneity of a grid region's formulas in order to take advantage of the approach described in this paper, but this knowledge is easily obtained. It is already present in those spreadsheet languages in which the user is allowed to explicitly share a single formula among several cells (e.g., Lotus™, Forms/3 [3, 4, 5], Formulate [31], Prograph spreadsheets [29], and Chi et al.'s visualization spreadsheet language [7]). If not already present, it can easily be gathered "behind the scenes" by a spreadsheet system, such as by maintaining knowledge of the relationships among copied formulas as in [10].

2.2 Static grids versus dynamic grids

There are two attributes of grids and regions that are static in some spreadsheet languages and dynamic in others, and these attributes significantly impact the manner in which testedness of grid cells can be tracked. The first is whether a grid's size (number of rows and columns) is specified statically or dynamically. Static specification of grid size is the norm for commercial spreadsheet systems, but some research systems use dynamic size specifications (e.g., Forms/3 and Formulate).

The second of these two attributes is whether determination is static or dynamic as to exactly which cells are being referenced in a formula. The most common approach in commercial spreadsheet systems is static, restricting cell row/column references to be based only on static position, optionally offset by a constant.

Traditional imperative languages—for which most research in testing has occurred—typically support statically-sized, dynamically-referenced grids via arrays. Approaches for reasoning about the testedness of array elements have been suggested [12, 14, 15]; in general, however, the problem of precisely treating array references at the element level is unsolvable for

the dynamic referencing that is the norm in imperative programs. Thus, the prevalence of static referencing in the spreadsheet paradigm affords unusual opportunities for reasoning about testedness.

In summary, for viable application to commercial spreadsheet systems, a testing methodology must at least support statically-sized, statically-referenced grids. The two approaches described in this paper do support this type of grid, and also support the dynamically-sized, statically-referenced grid type.

2.3 Grids in Forms/3

Our work was prototyped using Forms/3 grids. In Forms/3, a *grid* is a tuple (*ID, region set, row dimension cell, column dimension cell*), where the *row dimension cell* and *column dimension cell* are two distinguished cells whose formulas define the grid's dimensions, and the *region set* is a set of regions. Each *region* is also a tuple (*ID, cell set, formula*), where the *cell set* is the set of all the cells contained in the region (each element of which is termed an *element cell*), and *formula* is an expression shared by all element cells. Formula syntax follows conventional spreadsheet syntax, with the addition of "pseudo-constants" *i* and *j* to mean "this row" and "this column," respectively.

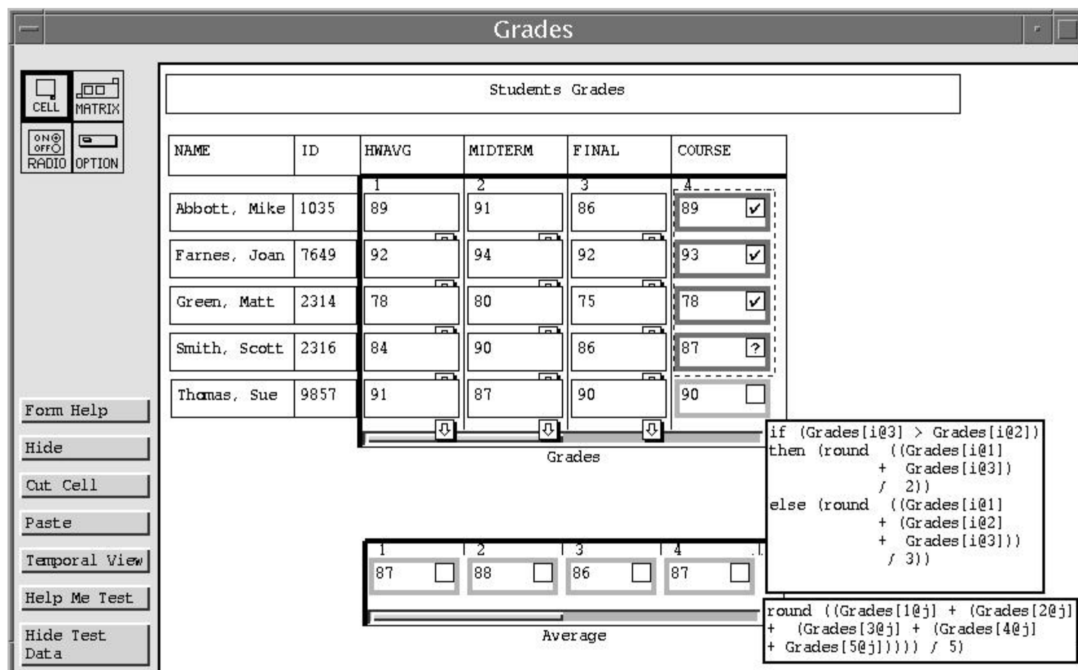


Figure 2. A version of the `Grades` spreadsheet using Forms/3 grids under the Straightforward approach. The user can enter a formula via a formula tab (⇩). The input cells each have their own formulas (one cell per region), but note that the rightmost column (region) has a single shared formula, as does the `Average` grid. The user is in the process of selecting four `COURSE` cells by stretching the dotted rectangle.

To define values for a Forms/3 grid's cells, the user statically partitions the grid into rectangular regions and, for each region, enters a single formula for all element cells it contains. To statically derive a cell's formula from its shared region formula, the system replaces any pseudo-constants i and j in the formula by the cell's actual row and column number. The row dimension cell and column dimension cell can have arbitrarily complex formulas. Figure 2 shows a spreadsheet similar to the spreadsheet shown in Figure 1, but rewritten to use grids. The row and column dimension formulas (not shown) are simply constants in this example.

2.4 The cell relation graph model

In our previous work [25, 26] we defined an abstract model for spreadsheets, called a *cell relation graph* (CRG), that we use to model those spreadsheets and to define and support testing. The approaches described here for testing grids are based upon this model. A CRG is a pair (V, E), where V is a set of *formula graphs*, and E is a set of directed edges called *cell dependence edges* connecting pairs of elements in V. Each formula graph in V represents the formula for a cell, and each edge in E models the data dependencies between a pair of cells. There is one formula graph for each cell in the spreadsheet. Each formula graph models flow of control within a cell's formula, and is comparable to a control flow graph representing a procedure in an imperative program [1, 22]. Thus, a formula graph is a set of nodes and edges. The nodes in a formula graph consist of an *entry node* modeling initiation of the associated formula's execution, an *exit node* modeling termination of that formula's execution, and one or more *predicate nodes* and/or *computation nodes*, modeling execution of `if`-expressions' predicate tests and all other computational expressions, respectively. The edges in a formula graph model control flow between pairs of formula graph nodes. Edges that are out-edges from predicate nodes are labeled with the value to which the conditional expression in the associated predicate must evaluate for that particular edge to be taken.

For example, Figure 3 depicts a portion of the CRG for the simple spreadsheet shown in Figure 2. Each formula graph is delimited by a dotted rectangle. In the figure, formula graph nodes labeled E and X are entry and exit nodes, respectively. Nodes with multiple out-edges (represented as rectangles) are predicate nodes. Other nodes are computation nodes.

2.5 The du-adequacy criterion for spreadsheets

Using the CRG model, we defined a test adequacy criterion for spreadsheets, which we refer to as the *du-adequacy criterion.* We summarize it somewhat informally here; a full formal treatment has been provided elsewhere [25].

The du-adequacy criterion is a type of dataflow adequacy criterion [11, 12, 16, 22]. Such criteria relate test adequacy to interactions between definitions and uses of variables in source code (*definition-use associations*, abbreviated *du-associations*). In spreadsheets, cells play the role of variables; a *definition* of cell *C* is a node in the formula graph for *C* representing an expression that defines *C*, and a *use* of cell *C* is either a *computational use* (a non-predicate node that refers to *C*) or a *predicate use* (an out-edge from a predicate node that refers to *C*). Under the du-adequacy criterion, cell *C* is said to have been *adequately tested* (*covered*) when all of the du-

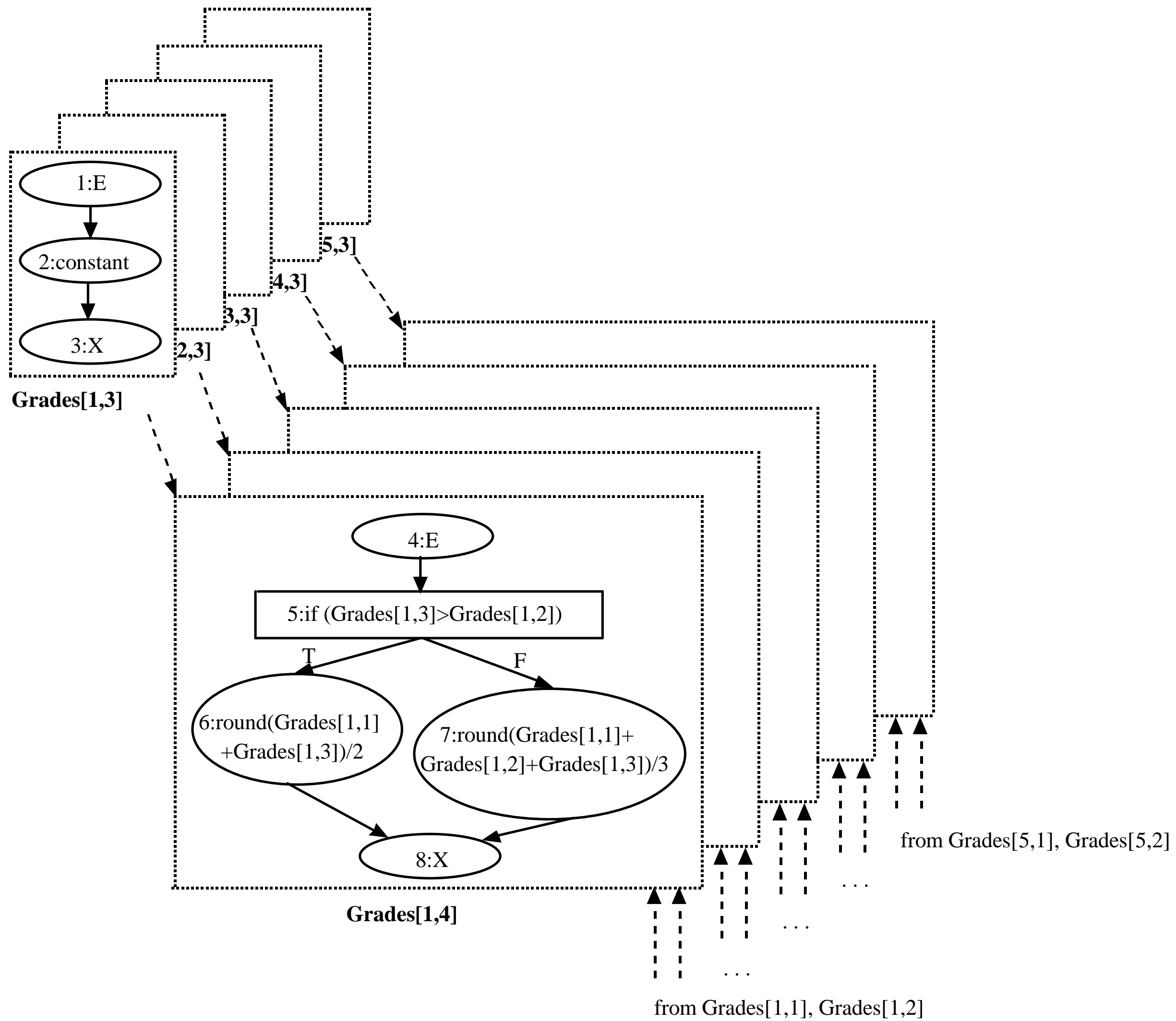


Figure 3. A portion of the CRG for the spreadsheet of Figure 2. Shown are formula graphs for the `Grades` grid.

associations whose uses occur in *C* have been *exercised* by at least one test: that is, where inputs have been found that cause the expressions associated with both the definitions and uses to be executed, and where this execution produces a value in some cell that is pronounced “correct” by a user validation. (The closest analogue to this criterion in the literature on testing imperative programs is the “output-influencing-All-du” dataflow adequacy criterion [11], a variant of the “all-uses” criterion [22].) In this model, a *test* is a user decision as to whether a particular cell contains the correct value, given the input cells’ values upon which it depends.

For example, the du-associations involving Abbott’s `FINAL` cell (`Grades[1,3]`) and his `COURSE` cell (`Grades[1,4]`), using the node numbers shown in Figure 3, are (2,5T), (2,5F), (2,6), and (2,7). Hence, under the du-adequacy criterion, `Grades[1,4]` is adequately tested when there has been a test in which `Grades[1,3]` was greater than `Grades[1,2]`, exercising du-associations (2,5T) and (2,6), and another test in which `Grades[1,3]` was not greater than `Grades[1,2]`, exercising du-associations (2,5F) and (2,7). (We are simplifying this discussion by ignoring the uses of Abbott’s `MIDTERM` and `HWAVG`, since their formula graphs are not explicitly shown in Figure 3.)

3. The Straightforward approach

An obvious approach to explicitly supporting grid testing is to enhance the user interface so that the user can validate all or part of an entire region in one operation, but to have the system maintain testedness information about each cell individually using the CRG model just as described above. We term this approach the *Straightforward* approach. The Straightforward approach modifies our previous methodology’s algorithms in straightforward ways to facilitate working with large grids. It is important to consider even such a simple strategy seriously, because if it is viable, there is no reason to invest in more elaborate strategies.

In the Straightforward approach, the only change from the user’s perspective is that a group selection device such as a rubberband is added. The user can use this device to select a group of cells in a grid and validate any of the selected cells, which applies the validation to all the selected cells. The rubberband does not “declare” any permanent relationship; it is simply a transient selection device. When the user does not use the rubberband, the user’s validation of one grid cell *X* applies to *X* and only *X*, just as in the previous methodology. Other than the rubberband, the visual communication devices are exactly the same as in the original WYSIWYT methodology. Thus, in Figure 2, every cell has its own testing color border and validation checkbox, just as in our earlier work. For example, if no cells in Figure 2 had been

validated yet and then the user selected and validated Abbott's `COURSE` cell only, which executes the predicate and the `else`-expression in the formula, then only Abbott's `COURSE` cell would have been shown in purple (medium gray), denoting partially tested.

A strength of the Straightforward approach is that, because all information is kept individually for each cell, the user has the flexibility to validate *any* arbitrary group of contiguous[1] cells, or even any cell individually. For example, the actual scenario of Figure 2 is that the user somehow previously validated the top four `COURSE` cells, and then changed the input contributing to the fourth (Smith's) cell, which is why its validation checkbox now contains a "?". At the point the figure was captured, the user was in the process of rubberbanding the top four `COURSE` cells in order to validate that group all at once, since all of those cells use the `else` part of the formula and the value being scrutinized in Smith's `COURSE` cell is correct. In this scenario, the user plans to next attend individually to Thomas's `COURSE` cell, which uses the `then` part.

3.1 Information required by the approach

As in the original WYSIWYT methodology, the Straightforward approach requires the information described in Table 1 for each cell. Like other spreadsheet languages, our system can retrieve or update any cell efficiently, accomplished via a hash table in our system.

	Information collected	Description	When collected
1	*C.DirectProducers*	The cells referenced explicitly in *C*'s formula.	Statically
2	*C.DirectConsumers*	The cells whose formulas explicitly reference *C*.	Statically
3	*C.Defs*	The definitions explicitly present in *C*'s formula.	Statically
4	*C.Uses*	The uses explicitly present in *C*'s formula.	Statically
5	*C.DUAs:* *C.DUAs.Incoming* *C.DUAs.Outgoing* *C.DUA:* *C.DUA.definition* *C.DUA.use* *C.DUA.exercised*	A set of du-associations, consisting of: All du-associations whose uses are in *C.Uses*. All du-associations whose definitions are in *C.Defs*. An element of *C.DUAs*, in format (definition, use, exercised), consisting of: The definition. The use. True if *C.DUA* has been exercised; otherwise false.	 Statically Statically Dynamically
6	*C.Trace*	The set of *C*'s formula graph nodes that were executed in the most recent evaluation of *C*.	Dynamically

Table 1. The six types of primary information collected in the Straightforward approach for each cell C. They are collected by updating hash tables while parsing formulas (statically) and while executing formulas (dynamically).

[1] The mention of contiguity is *only* because the user interface device (a rubberband) needs to spatially surround the cells.

It is reasonable to rely upon the formula parser to keep the first four items in Table 1 up-to-date, because the first two are already needed to support the usual spreadsheet abilities of efficiently updating the screen and cached values after each formula edit, and the next two are easily collected while collecting the first two. The algorithms for maintaining the remaining items are described next.

3.2 The algorithms and their complexities

To support the testing of grids under the Straightforward approach, the system needs to perform four tasks. In describing cell relationships in these tasks, we use producer/consumer terminology. A *direct producer* of cell *C* is a cell referenced explicitly in *C*'s formula. From this, we recursively define a *producer* of *C* as either a direct producer of *C* or a direct producer of a producer of *C*. Similarly, a *direct consumer* of cell *C* is a cell that refers explicitly to *C* in its formula, and a *consumer* of *C* is either a direct consumer of *C* or a direct consumer of a consumer of *C*.

The four tasks for incrementally updating all the necessary information are:

Task 1: Collecting static information. Whenever the user edits a formula for *C*'s region, *C.DUAs.Incoming* and *C.DUAs.Outgoing* are re-collected. In addition, the outgoing and incoming du-associations in *C*'s direct producers and direct consumers, respectively, are updated.

Task 2: Tracking execution. Whenever *C* is executed, the most recent set of *C*'s formula graph nodes executed (*C*'s *trace*) is stored in *C.Trace*.

Task 3: Validation. Whenever the user validates *C* by clicking on it, each element of *C.DUAs.Incoming* whose use node is in *C.Trace* is marked exercised. This process is performed on each producer of *C* as well.

Task 4: Adjusting test adequacy information. Whenever the user edits some non-input cell's formula[1] for any producer *P* of *C*, *C.DUAs.Incoming*'s elements that directly or transitively contain uses of *P* are marked "not exercised".

For example, the result of Task 1, gathering du-associations, for cell `Grades[1,4]` in Figure 2 would include (2,5T), (2,5F), (2,6), and (2,7), as was discussed in Section 2.5; the result of Task 2, tracing its execution, would be {4,5,7,8}; the result of Task 3, validating it, would be that du-associations (2,5F) and (2,7) as well as some involving `Grades[1,1]` and

---

[1] A simple variant on Task 4 is triggered by editing an input cell's formula, which does not change testedness information, although it does change validation checkmarks of affected cells to question marks. This variant is not covered in detail in this paper, since it is simply a matter of omitting most of the work of the Task 4 algorithms.

`Grades[1,2]` would be marked "exercised"; and the result of Task 4, adjusting testedness after an edit, would be that such marks in its consumer, `Average[1,4]`, would be removed and the borders of both `Grades[1,4]` and `Average[1,4]` would be reset to the untested color (red or light gray). Note that, because these tasks are not triggered by the same user actions, they run at different times.

Figure 4 gives the algorithm for Task 1, collection of du-associations. Whenever the user edits the region's formula, this algorithm processes each cell in the region. After deleting prior information (using `deletePriorDUAInformation`, an algorithm of the same structure and complexity as the algorithm in Figure 4), the main part of the algorithm proceeds. Note the use of `StaticallyResolve`; it is an O(1) routine that returns the actual cell to which a reference with relative indices resolves. For example, if some cell *M[1,3]*'s formula contains a reference to *P[i,j-1]*, then `StaticallyResolve`*(M[1,3],P[i,j-1])* returns *P[1,2]*. The low cost of `StaticallyResolve` depends upon the static referencing common in spreadsheet languages and on the ability to determine statically which of the shared formulas is the appropriate one for a given cell. The mechanisms for formula sharing in commercial spreadsheet systems are static, so they fulfill this requirement. The formula sharing mechanism in Forms/3 is also static, using regions, so it too fulfills this requirement. Given static region sizes, `StaticallyResolve` works even in the case of dynamically-sized grids, because in that combination each region size except one (the one that has been defined to hold elements not in any other region) still has a statically-determined size, which means the applicable region (correct shared formula) for any

```
algorithm CollectAssocSF(R)
  for each cell C ∈ R do
    CollectAssoc(C)
```

```
algorithm CollectAssoc(C)
  deletePriorDUAInformation(C)
  for each cell DP in a use ∈ C.Uses do    //direct producers
    if DP is a grid cell reference then
      DP = StaticallyResolve(C, DP)
    for each definition ∈ DP.Defs do
      let DUA = (definition, use, false)
      add DUA to C.DUAs.Incoming
      add DUA to DP.DUAs.Outgoing
  for each use of a definition ∈ C.Defs do //direct consumers
    let DC be the cell containing the use
    let DUA = (definition, use, false)
    add DUA to C.DUAs.Outgoing
    add DUA to DC.DUAs.Incoming
```

Figure 4. Straightforward approach Task 1 algorithm for collecting a region's du-associations for region *R*.

cell can be statically determined. `StaticallyResolve` is needed only in the producer loop, in processing the formula to determine *C*'s relationships to its direct producers; it was already performed earlier for *C*'s consumer cells, namely at the times their own formulas were edited.

Except for its introduction of `StaticallyResolve`, `CollectAssoc` is simply a slightly modified version of the algorithm developed for the original WYSIWYT methodology. It is called *n* times by `CollectAssocSF`, once for each cell in a region of size *n*. Thus, because `CollectAssoc` is called for every cell in the region, its cost is *n* times the single-cell cost, a cost incurred whenever the region's formula is edited. The single-cell cost is $O(p_d + c_d)$, where p_d and c_d are *C*'s number of direct producers and direct consumers, respectively, assuming a constant-bounded formula length [25]. Thus, for a region of size *n*, the total cost is $O(n(p_d+c_d))$ for `CollectAssocSF`, where p_d and c_d have the definitions above generalized to support the context of regions: they are the number of direct producers and direct consumers, respectively, of the worst-case cell in the region.

Task 2, collecting each cell's execution trace, must be performed whenever a cell executes, and is accomplished simply via a probe in the evaluation engine. Thus, this task can be done easily and efficiently, adding only O(1) to the cost of executing a cell, and is incurred only for the cells that actually execute.

For Task 3, the Straightforward approach marks "exercised" the cell's relevant du-associations, as well as those of its producers (i.e., its backward dynamic slice), and shows via colors the resulting increase in du-associations exercised by the user's testing. To do so, it simply calls the original WYSIWYT methodology's version of `ValidateCoverage` *n* times, where *n* is the number of cells in the selected group of cells. `ValidateCoverage` was presented in our earlier work, and its time complexity is O(*p*), where *p* is the number of *C*'s producers [25, 27]. It sets *C.DUA.exercised* to true for each element of *C.DUAs.Incoming* whose use is in *C.Trace*, and then recursively calls `ValidateCoverage` on each cell referred to in *C.Trace*'s uses. Finally, it updates the screen display of *C* to reflect its new testedness status. To facilitate comparison with the approach presented in the next section, we assume the selected group is a region of *n* cells, which is what the user would select in order to validate all element cells sharing the region's formula. Since the cost of this task is *n* times the cost of the task performed on a single cell, the Task 3 total is O(*np*), where *p* represents the number of producers (worst case) of any cell in the selected group.

For Task 4, whenever a region's formula is edited, the Straightforward approach repeatedly calls `AdjustTestedness`, the original WYSIWYT methodology's version [25, 27], for every cell in the region. Once invoked on *C*, `AdjustTestedness` keeps calling itself until every

consumer of C has been processed. Thus the cost of the task is n times the cost of the task performed on a single cell, or O(nc), where c is the number of consumers (worst case) of any cell in the region.

4. Region Representative approach

The *Region Representative* approach is a more elaborate approach. It aims directly at Problem 2 (system efficiency) by doing most of its reasoning at the granularity of entire regions rather than at the granularity of individual cells, thereby removing some of the dependency on region size. (Recall from Sections 2.1 and 2.3 that a region is a group of adjacent cells known by the system to have been given the same formula by the user; an example is Figure 5's `COURSE` column, whose cells share the formula shown.) Unlike in the Straightforward approach, in the Region Representative approach the user does not explicitly select a group of cells to validate; rather, the user's validation of a single cell is considered to be a validation of all cells in the same region for the same input values as those affecting the explicitly validated cell. This approach improves system efficiency over the Straightforward approach and provides many conveniences to the user, some of which are even greater than in the Straightforward approach, but it does not provide quite as much flexibility to the user.

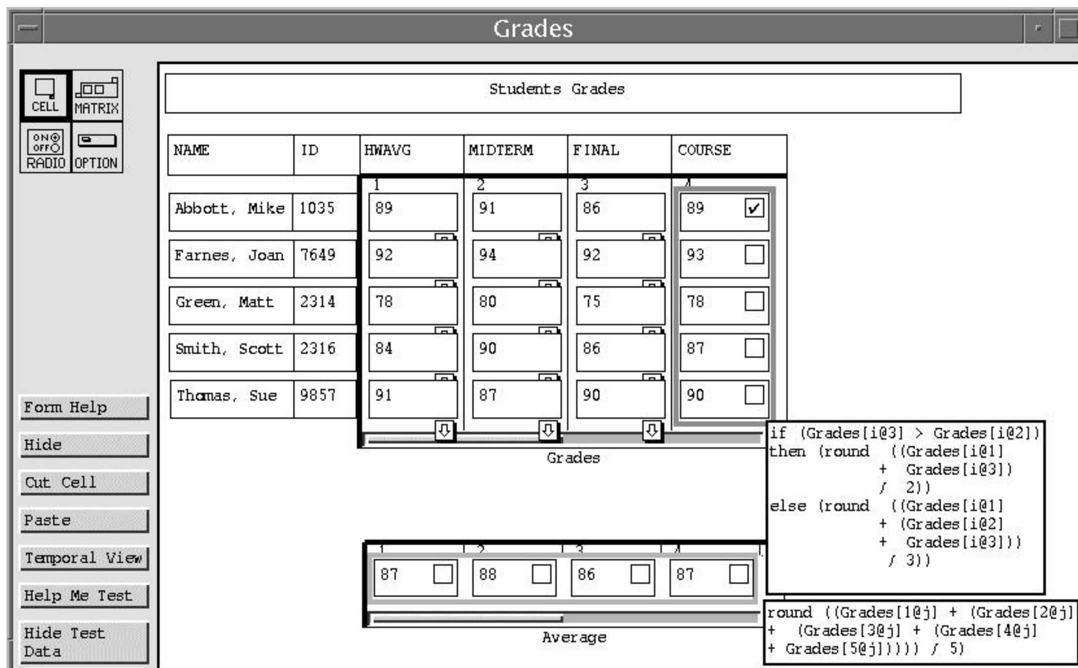


Figure 5. The `Grades` spreadsheet from Figure 2 shown here under the Region Representative approach. The user is in the process of testing by clicking the validation checkbox of Abbott's `COURSE` cell, which validates du-associations connected with the `else` part of the formula for the entire `COURSE` region.

The visual devices depict the reasoning differences from the Straightforward approach. The crux of these differences is that the information collected when a user validates an element cell is shared with all the other elements in the region, and this sharing is indicated via a single testing border around the entire region, as in Figure 5. For example, if no cells in Figure 5 have been validated yet and then the user validates Abbott's `COURSE` cell, which executes the predicate and the `else`-expression in the formula, the `COURSE` column's testing border turns purple (medium gray), which is the point at which the screenshot in Figure 5 was made. If the user subsequently validated Thomas's `COURSE` cell, which executes the `then`-expression, the entire column's testing border would then become blue (black), denoting fully tested.

Although an important motivation in developing the Region Representative approach has been to reduce the workload of the system, the Region Representative approach also offers several advantages to the workload of the user. These advantages stem from the fact that the user does less test input generation manually: a large grid already provides a variety of input data. The first advantage is that the user may not need to conjure up new test inputs. For example, in the `Grades` spreadsheet, the user tested Abbott's `COURSE` cell in part by selecting another cell for validation—Thomas's `COURSE` cell—because it had a useful set of test inputs already contributing to it. In contrast to this, in the Straightforward approach the user could achieve coverage on Abbott's `COURSE` cell only by editing in different input values to force execution of both branches in *that particular* cell (Abbott's). This leads to a mechanical advantage as well: the Region Representative approach requires fewer physical actions, i.e. edits and validation clicks, to achieve full coverage. This mechanical advantage becomes significant in a large grid, such as a 500-student version of the `Grades` spreadsheet. The third advantage is that, when the user does not provide a new test input, he or she does not need to modify the "real" input data and then remember to restore it. Fourth, the user's job as oracle (decider of the correctness of values) may be easier with the Region Representative approach, because with so many inputs from which to choose, it may be possible to notice one that produces obvious answers, such as Thomas's values (bottom row of Figure 5).

An apparent disadvantage is loss of flexibility: the user seems to have no way to prevent the propagation of testedness to all the cells in the region. Hence, some functionality is lost. For example, the user cannot exclude a cell from group tests in favor of individualized testing, such as a cell that refers to an out-of-range value. However, most instances of this disadvantage can be removed by allowing the user to subdivide a region into more than one region for testing purposes. For example, suppose there is a region *R* in which each cell is computed by adding one to the cell above it. The user might want to test the top row of a rectangular region

separately because it is based on an initial set of values (those provided by a different region above it) rather than upon cells in the same region. To do this in our prototype, the user simply subdivides *R* into two regions, *R1* and *R2*, and tests them separately.

4.1 CRG model modifications

The Region Representative approach requires modifications to the CRG model described in Section 2.4. The first modification is that, instead of a formula graph for each cell in a region *R*, *R*'s cells are collectively modeled by a single formula graph of an abstract cell *Rij*, termed the *region representative* for region *R*, such as in Figure 6. The second modification is that du-associations are separated into two classes: those whose definitions occur in input cells (termed *constant du-associations*) and those whose definitions do not. Consider *CS*, a set of constant du-associations with the same use; all cells whose du-associations' definitions are elements of *CS* are said to be members of the same *constant region*. Each constant region is represented by a single region representative, since all of its members are in essence simply different input possibilities for the same use.

Using a single formula graph to represent multiple constant definitions involving the same constant use is important to the practicality of the approach. Without this device, a user would

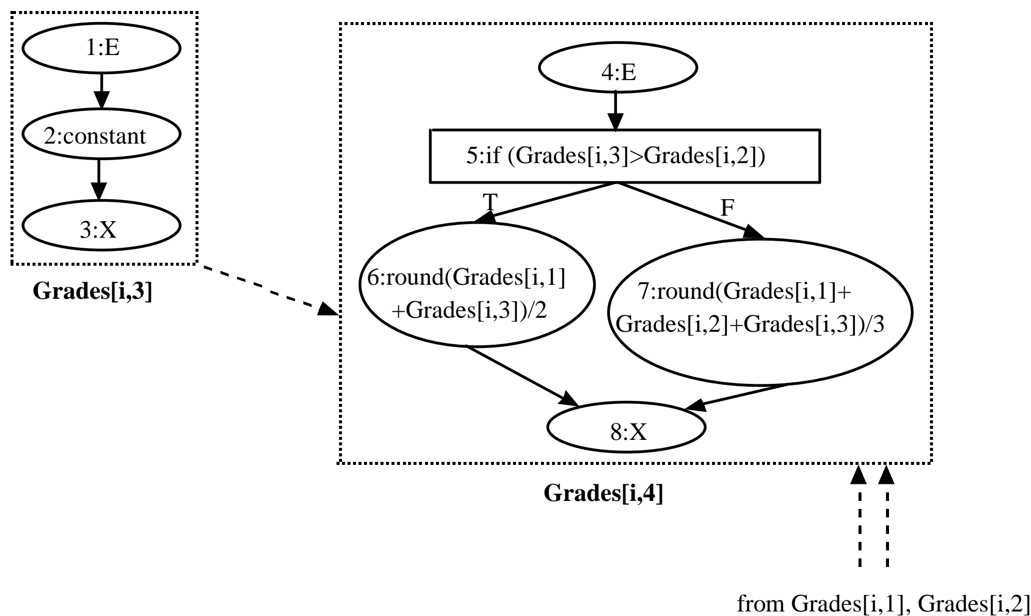


Figure 6. CRG showing the region representative of Figure 5's `Grades`' column 4, labeled *Grades[i,4]* here. Column 3 contains input cells that do not have any shared formulas, but form a constant region, labeled *Grades[i,3]*. Constant regions are also formed by columns 1 and 2. Note how much smaller this CRG is than the Straightforward approach's version of Figure 3: there are 4 CRG nodes in this figure, as compared to the 20 nodes in Figure 3.

have to validate enough cells to involve every input cell in the spreadsheet—one for each student, in the `Grades` example—and this would interfere with the scalability needed to solve Problem 2. For example, returning to Figure 5, all cells in the `Grades` grid's columns 1-3 (labeled `HWAVG`, `MIDTERM`, and `FINAL`) have individual constant formulas. Since there are five students, there are five constant du-associations terminating at the same use: the reference to `Grades[i@3]` in the `COURSE` region's `then`-expression. If the system required each of these du-associations to be validated separately, the user would need to validate five rows twice, modifying inputs for each, to exercise all the references in the `then` and the `else` referring to these separate uses. However, since constant du-associations terminating in the same use node are represented by a single representative instead, as illustrated in Figure 6, the user can pick just one row that has been executed by the `then` case (Abbott's row in our example) and thereby exercise all the region's du-associations involving that use node with just one test, and then pick one more row (e.g., Thomas's) for the `else` case.

4.2 Information collected

To realize the Region Representative approach, most of the testing-oriented information corresponding to that described in Table 1 must be shared among element cells of a region via the region's representative. See Table 2.

	Information collected	Description
1	*Rij.DirectProducers*	The region representatives and non-region cells that are potentially referenced in *Rij*'s formula.
2-5	*Rij.DirectConsumers,* *Rij.Defs, Rij.Uses,* *Rij.DUAs* and its components	Same as Table 1, except replacing "*C*" by "*Rij*", and replacing "cells" by "region representatives and non-region cells". For example, item 2 becomes: The region representatives and non-region cells whose formulas explicitly reference *Rij*.
6	*C.Trace*	The set of *Rij*'s formula graph nodes that were executed in the most recent evaluation of cell *C*.

Table 2. Region Representative approach's information collected. In the Region Representative approach, only the trace information is still stored for each cell. The region representative *Rij* stores the rest of the testing-oriented information as a representative of all cells in its region. (The numbering at left refers to the numbering of Table 1.)

4.3 Task 1 reasoning: Collecting static information for region *R*

The algorithm for collecting *Rij.DUAs* (Task 1) is shown in Figure 7. As in the Straightforward approach, this algorithm is triggered whenever a formula is edited. One important difference from `CollectAssocSF` (Figure 4) is that du-association collection is done once per region rather than once per cell. The other important difference is that if region

representative *Rij* refers to grid *G*'s cell *DP*, then the representative of every region in *G* to which *DP* could possibly belong must be regarded as a source of definitions used by *Rij*.

`CollectAssocRR` uses `StaticallyResolveRegion`, which is similar to `StaticallyResolve`, but returns information about regions rather than about cells. Given a region *R* and its representative *Rij* whose formula includes a reference *P[i,j-1]*, `StaticallyResolveRegion` returns a list of representatives for regions to which *P[i,j-1]* could belong, at a cost of O(*r*) where *r* is the number of regions in grid *P*. In `StaticallyResolveRegion`, *Rij* provides the context. For example, if *R*, which includes row 1 from columns 2 to 4, refers to *P[i,j-1]*, where *P* is a grid with regions at the same positions as in *R*'s grid, then `StaticallyResolveRegion(`*Rij,P[i,j-1]*`)` returns two representatives: one for the region of *P* containing only row 1 column 1, and one for the region of *P* containing row 1 columns 2 to 4.

`StaticallyResolveRegion`'s reasonable time cost is possible because regions are rectangular and contiguous; hence geometric reasoning can be used instead of a search to determine whether a cell reference could be within a particular region's boundaries. In the above example with *Rij* as the representative for region *R* spanning (1,2) to (1,4), *Rij*'s reference to *P[i,j-1]* falls within the rectangle bounded by (1,1) and (1,3); hence, any region in *P* overlapping this rectangle is potentially the region in which one of the *P[i,j-1]*s actually resides.

It is also possible to extend `StaticallyResolveRegion` to accommodate non-contiguous and non-rectangular regions. For example, in the `Grades` spreadsheet of Figure 5,

```
algorithm CollectAssocRR(Rij)
  deletePriorDUAInformation(Rij)
  for each cell/rep DP in a use ∈ Rij.Uses do    //direct producers
    if DP is an ordinary cell
      then regReps = {DP}
      else regReps = StaticallyResolveRegion(Rij,DP)
    for each defRij ∈ regReps do
      for each definition ∈ defRij.Defs do
        let DUA = (definition, use, false)
        add DUA to Rij.DUAs.Incoming
        add DUA to defRij.DUAs.Outgoing
  for each use of a definition ∈ Rij.Defs do     //direct consumers
    let DC be the cell/rep containing the use
    let DUA = (definition, use, false)
    add DUA to Rij.DUAs.Outgoing
    add DUA to DC.DUAs.Incoming
```

Figure 7. Task 1 algorithm for collecting a region's du-associations in the Region Representative approach.

suppose Green has a medical emergency and misses the midterm exam, and suppose the instructor chooses to calculate Green's `COURSE` grade using a special formula that omits the `MIDTERM` cell. Geometrically, this would divide the `Grades` grid's `COURSE` column (region) into three rectangular regions—one for every student above Green, one for Green's `COURSE` grade, and one for every student below Green—but logically, two regions would be more appropriate because only two distinct formulas are needed—one for Green's `COURSE` grade, and one for all the other students' `COURSE` grades. Supporting a region consisting of multiple disjoint rectangles could be done if `StaticallyResolveRegion` (and `StaticallyResolve`) were changed to reason about each rectangle rather than about each region, changing the cost to O(*rect*), where *rect* is the number of rectangles. Since ignoring non-rectangular and non-contiguous regions in this presentation does not result in loss of generality, for the rest of this paper we will ignore them for brevity.

4.3.1 Time complexity of Task 1

Suppose there is a region *R* and its representative is *Rij*. Let p_d' be the number of region representatives[1] that are potentially referenced by *Rij*'s uses, i.e., a conservative (static) definition of *Rij*'s direct producers. This is the set traversed by the first two loops of the algorithm. Within these first two loops is a call to `StaticallyResolveRegion`, which costs O(*r*), where *r* is the maximum number of regions in a grid being referred to by any one reference in *Rij*'s formula. Let *f* be the maximum number of definition nodes in any referenced region's formula graph. This is the set traversed by the innermost loop. Finally, let c_d' be the number of region representatives that have previously been determined to potentially reference *Rij* (a conservative definition of *Rij*'s direct consumers). These are the cells visited by the last loop. Thus, the total time cost of the algorithm is:

$$O(rfp_d'+c_d')$$

The above cost can be further simplified when there is a maximum imposed on formula lengths. Most spreadsheet languages have such a maximum; for example, in Excel, the maximum is 1024 characters. Given the presence of such a maximum, *f* becomes constant-bounded by the maximum formula length, simplifying the asymptotic time cost of Task 1 to:

$$O(rp_d' + c_d')$$

[1] For simplicity, in cost analyses we will consider a cell that is not in a region to be a representative of itself.

as compared to the Straightforward approach's cost of O(np_d+nc_d). The savings over the Straightforward approach's cost come from all three factors. This is because *Rij*'s p_d' direct producers and c_d' direct consumers include region representatives, each of which potentially replaces multiple elements in all the element cells' p_d direct producers and c_d direct consumers. Similarly r is potentially much smaller than n. For example, for the `COURSE` region of `Grades` (i.e., `Grades[i,4]`) of Figure 5 and Figure 6, although c_d' and c_d are equal (c_d'=c_d=1), and p_d' and p_d are equal (p_d'= p_d=3), r=1 region whereas n=5 cells in the `COURSE` region.

4.3.2 Cost of Task 1 in context

This algorithm is triggered when the user edits region *R*'s formula. At this point, the costs any spreadsheet system must incur even without the existence of a testing subsystem are those of parsing the formula, which costs at least the number of characters in the formula; of calculating at least the on-screen cells in *R*, requiring visits to the on-screen cells in *R* and some of their producers; and of notifying consumers of the edited cell that their values are out of date, requiring recalculation and/or discarding of any previously cached values [25].

Of these three costs, the costs of evaluation and notification are the most useful to consider, because they are the greatest that involve the same cell sets as in Task 1. If all of the region's cells are on the screen, the cost of evaluation is at least as great as O(p_d'), because the system needs to revisit at least all direct producers and to recalculate producers (including direct producers) that do not already have up-to-date cached values. (Each cell can keep a pointer to its representative, as is the case in our implementation, so that a visit to a cell can lead to the representative with only a constant cost addition.) Notification of consumers requires the system to visit at least all the direct consumers. Since evaluation and notification visit at least the same cells as in Task 1, then when the region is entirely on the screen and r is small—which is the case for spreadsheets with many shared formulas and/or multiple input cells—the cost of Task 1 in the context in which it is performed increases by a multiplicative constant factor. However, when these conditions do not hold, whether the worst case cost of Task 1 increases by more than a multiplicative constant factor the cost of work triggered by a formula edit depends on the host spreadsheet language's particular evaluation and caching strategies.

4.4 Task 2 reasoning: Tracking execution traces

Storing execution traces whenever a cell executes is, as in the Straightforward approach, implemented simply by inserting a probe into the evaluation engine, so no separate algorithm is presented here. The probe adds O(1) to the cost of evaluating a cell.

4.5 Task 3 reasoning: Validating all of region *R*

The algorithm for validation is shown in Figure 8. `ValidateRepRR` is similar to a *single* call to the original WYSIWYT methodology's `ValidateCoverage` algorithm, with the difference that `ValidateRepRR` derives cell references from generic region row/column specifiers, which is accomplished via the call to `StaticallyResolve`. Even though trace information for each cell is used, in general this algorithm reasons at the granularity of regions about du-associations in the region's formula. It marks as exercised region du-associations involving uses in the cell's trace being validated, and then repeats recursively on the producers contributing to these uses. In contrast to this, in the Straightforward approach, Task 3 makes *multiple* calls to the original WYSIWYT methodology's `ValidateCoverage` algorithm.

Suppose the algorithm's incoming parameter cell *C* exists in region *R*, whose representative is *Rij*. Let *p* be the worst-case number of producers of any element cell in *R* (i.e., the worst-case size of the backwards dynamic slice of any cell in *R*). `ValidateRepRR` traverses du-associations for all direct producers in *C*'s trace (jointly accomplished by the top two loops), and then has a recursive call to the producers of these producers. Thus, the total time cost is simply:

$$O(p)$$

as compared to the Straightforward approach's cost of $O(np)$.

Task 3 is triggered when the user performs one validation of cell *C*. Its cost is similar to the cost of evaluating a cell when there are no relevant cached values, but unlike the evaluator, Task 3 does not need to access any consumers to notify them that their cached values are out of date.

```
algorithm ValidateRepRR(C,ValidatedID)
  let R = C's region
  let Rij = R's region representative
  Rij.ValidatedID = ValidatedID
  for each use ∈ C.Trace do
    for each DUA in Rij.DUAs.Incoming do
      if DUA.use = use then
          let defCell = the cell referenced in DUA.definition
          if defCell is a grid cell then StaticallyResolve(C,defCell))
          if DUA.definition ∈ defCell.Trace then
            DUA.exercised = true
            if defCell.ValidatedID < ValidatedID then
              if defCell is not a grid cell
                then ValidateCoverage(defCell,ValidatedID)
                else ValidateRepRR(defCell,ValidatedID)
  UpdateDisplay(R)
```

Figure 8. Task 3 algorithm for validating a cell in region *R* under the Region Representative approach.

```
algorithm AdjustTestednessRR(C,UnValidatedID)
  let R = C's region
  let Rij = R's region representative
  for each DUA ∈ Rij.DUAs.Outgoing
    DUA.exercised = false
    for each useCell referenced in DUA.use
      if useCell.UnValidatedID < UnValidatedID  then
        if useCell is an ordinary cell
          then AdjustTestedness(useCell,UnValidatedID)
          else AdjustTestednessRR(useCell,UnValidatedID)
  UpdateDisplay(R)
```

Figure 9. Task 4 algorithm to adjust testedness when a shared (region) formula is edited.

4.6 Task 4 reasoning: Adjusting testedness for region R

The algorithm for adjusting testedness is shown in Figure 9. It is called when a region's formula is edited. For each outgoing du-association in the region, it marks the du-association "not exercised," and recursively processes consumers that make use of the edited region.

The two explicit loops traverse the du-associations in direct consumers of *C*. There is also a recursive call to consumers of these consumers. The algorithm is quite similar to that of the Straightforward approach, except that it stops when it has visited all consumers' representatives (which is the same as the number of consumers for *one* cell), instead of visiting all consumers of *all* cells. Thus, the total time cost of Task 4 is:

$$O(c)$$

where c is the number of consumers' representatives referred to by *C*'s region representative's formula, as compared to the Straightforward approach's cost of $O(nc)$.

Task 4, like Task 1, is triggered when a new formula is entered for region *R*. Most evaluation strategies require visits to all the consumers of *R*'s cells for purposes of discarding cached values and/or recalculating them, the specifics of which depend on whether the engine uses lazy or eager evaluation [25]. Because of this fact, Task 4 under the Region Representative approach adds only $O(1)$ to the other work that is normally performed by a spreadsheet system without a testing subsystem.

4.7 A high-level overview of cost savings

The number of cells visited to reason about testing provides a high-level system-independent measure of time savings potential. Table 3 compares the two approaches on this basis.

System Task	Trigger	Cells Visited: Straightforward Approach (SF)	Cells Visited: Region Representative Approach (RR)
Task 1: Collect du's for region *R*.	The user changes region *R*'s formula.	*SF1* = |*R*'s direct producers| + |*R*'s direct consumers| + *n*	*RR1* = |*R*'s direct producers' representatives| + |*R*'s direct consumers' representatives| + 1
Task 2: Track execution traces.	1 or more cells execute.	*SF2* = Number of cells executing	*RR2* = *SF2*
Task 3: Validate all of region *R*.	The user performs one validation.	*SF3* = |*R*'s producers| + *n*	*RR3* = |*R*'s producers' representatives| + 1
Task 4: Adjust testedness for *R*.	Same as Task 1.	*SF4* = |*R*'s consumers| + *n*	*RR4* = |*R*'s consumers' representatives| + 1

Table 3. Number of cells visited in reasoning about region *R* containing *n* cells. (For simplicity of this table, we defined an "ordinary" non-region cell to be a representative of itself.)

5. Effects of the visual devices on time costs

There are three visual devices used in the original WYSIWYT methodology to communicate testedness to users about individual cells. Two of them—border colors and validation checkbox contents (checkmark, question mark, or blank)—have already been shown in the figures. Here we consider the effects of these two devices and a third device on the time costs.

The third device available to users, which has not been shown in the figures to this point, is optional dataflow arrows[1] colored with "testedness" status in the same manner as the border colors. Not only do these arrows show dataflow paths among cells; when formulas are showing, they also show the interactions between formula subexpressions and the testedness of each—in other words, each du-association's testedness status. Because these arrows extend visual feedback about testedness to the granularity of interactions among subexpressions, they provide information that can direct users to a testing action that will increase testedness. To users, this additional information seems to be almost as important as the border colors: in an empirical study of the original WYSIWYT methodology, 100% and 92% of the participants reported that the border colors and arrows, respectively, were helpful to their testing effort [28].

In the Straightforward approach, all three of the visual devices were employed unchanged for grids, just as described in the paragraphs above. This was also the case in our earliest prototypes of the Region Representative approach. However, the impacts of this naive decision on both usefulness and time costs were dramatic. From the user's perspective, the number of arrows leading in and out of the cells in just a single grid were sometimes so great, the screen became

[1] To avoid adding too much clutter, each cell's arrows are transient, and appear/disappear when the user clicks on the cell.

swamped with these arrows, rendering worthless their communication value to users. From the system's perspective, the time savings that accrued from the Region Representative approach's behind-the-scenes reasoning improvements were so overshadowed by the high visual update costs of re-coloring each cell border individually, the savings were obliterated. Thus, both aspects of scalability were lost.

Changing the visual devices in the Straightforward approach to solve these problems did not seem reasonable, since the very essence of that approach is its reasoning about individual cells, and this must be reflected in visual communications with the user. However, the Region

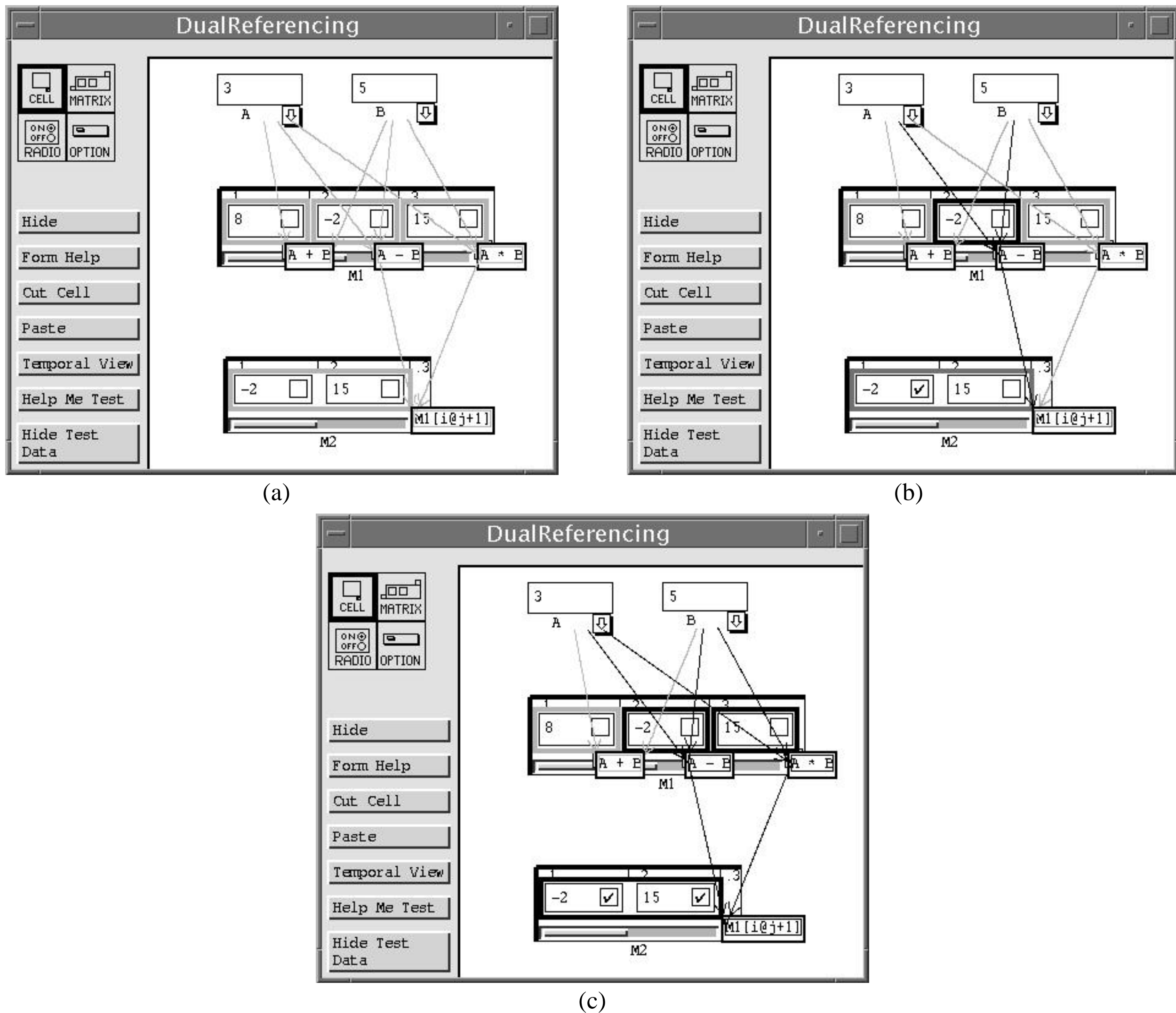


Figure 10. Since the Region Representative approach reasons at the granularity of regions rather than cells, borders and arrows depict testedness statuses and relationships at the granularity of regions as well (evident in grid `M2`). (a) The initial state of spreadsheet `DualReferencing` with the optional arrows showing. (b) The user has validated `M2[1,1]`; hence several borders and arrows become more blue (darker gray or black). (c) The user has now also validated `M2[1,2]`, which turns more of the borders and arrows blue (black).

Representative approach reasons region by region rather than cell by cell, and we were able to reflect this reasoning granularity in the visual devices. This can be seen by considering Figure 10. First, note that there is only one testing border, which surrounds the entire region in `M2`. Also, consider `M1[1,2]`'s definition, which `M2[1,1]` uses. This relationship is not depicted with arrows cell by cell, but rather region by region, by pointing into `M2`'s region's shared formula, or the region boundary if the formula is not showing. Thus, the arrows in the figure show that `M2`'s single region uses only the rightmost two of `M1`'s regions.

For a given spreadsheet S, let r be the number of regions in S and let n be the maximum number of element cells that any region has. Although Task 1 does not itself trigger visual updating, in order to support the later visual display of the colored arrows, Task 1 must collect du-associations about individual constant cells—in addition to the constant region representative du-associations that are sufficient for reasoning purposes. Doing so adds a cell-based loop to the Task 1 algorithm for the Region Representative approach, introducing a cost dependency on the number of constant cells (nr) rather than on the number of constant regions (r).

Task 3 and Task 4 entail visual updating, and these costs can be significant. Under the Straightforward approach, Task 3 requires repainting $O(nr)$ cells to update each cell's testing border. If colored arrows are on display, the cost increases to $O((nr)^2)$. However, under the Region Representative approach, this task requires only $O(r)$ updates of the testing borders, or $O(r^2)$ if colored arrows are displayed. For Task 4, the relationship between the Straightforward approach and the Region Representative approach is the same in terms of r and n as for Task 3, for border and arrow updating. In addition, Task 4 requires repainting the validation checkboxes of $O(nr)$ cells for both approaches.

The actual pixel repainting algorithms are part of the Garnet user interface toolkit [20], and incur the usual expense of reasoning about which pixels are visible and "dirty". Because these algorithms work with pixels, in addition to the dependency on the number of objects updated (expressed using r and n above) there is also a dependency on the physical sizes of these objects. Thus, repainting a single region's border is more expensive than repainting a single cell's border, which inherently adds some cost back to the Region Representative approach's region-based visual updating strategy. Such repainting algorithms are well established, and are not part of our work; hence they are not presented here.

6. Performance experiments

To provide additional, concrete information about how the scalability of the Region Representative approach compares to that of the Straightforward approach, we conducted performance experiments. The purpose of these experiments was to complement the analyses of the previous sections, which are about theoretically worst cases, with evidence about the approaches' actual time costs for large and small spreadsheets. Also, to measure the extent to which the effects of visual updating actually impact performance, the impact of costs associated with visual updating are considered.

Since the main objective of the experiments was to investigate scalability, we chose to vary only the size of the spreadsheet, holding other variables (such as degree of homogeneity, internal formula complexity, and interrelationships among formulas) constant. Figure 11 shows the spreadsheet used. In the figure, cells for 10 students are shown. The experiments were run on five different versions of this spreadsheet, involving 1, 10, 100, 200, and 500 students.

The spreadsheet used is similar to the simplified grade computation examples shown earlier in this paper, but contains a collection of formulas reflective of some common grading policies, such as allowing extra credit, rewarding improvement, and discarding the lowest quiz grade.

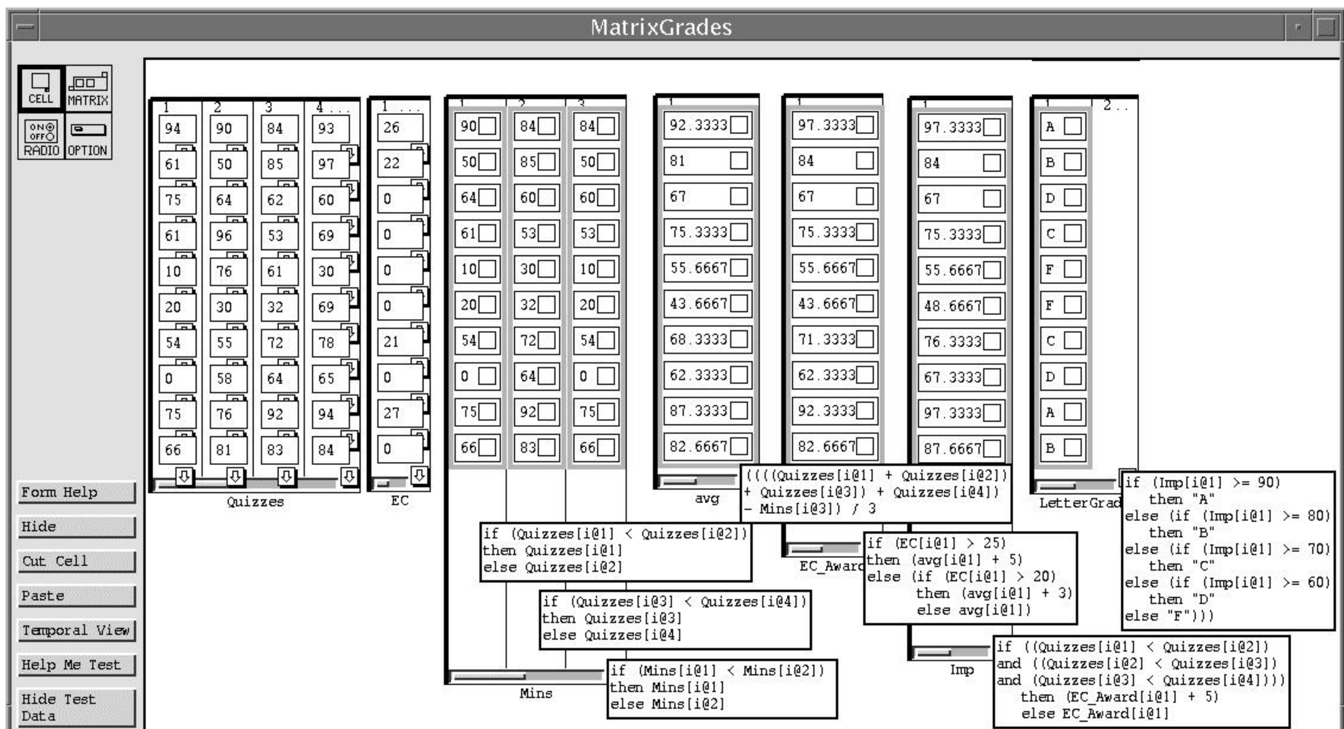


Figure 11. 10-student version of the `MatrixGrades` spreadsheet. (The bottom borders have been arranged to allow the relevant formulas to display without overlap.)

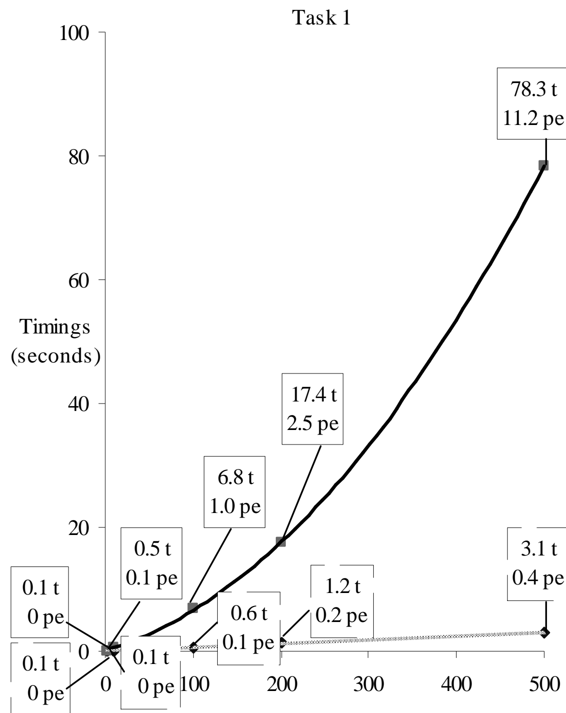
Also, for compactness on our screen shots and analyses, additional input cells that do not add materially to the computations, such as for student names and IDs, have been omitted. The spreadsheet computes a course letter grade (A, B, C, D, or F). The scores range from 0 to 100. The first two columns of `Mins` track the minimum of each pair of quiz scores, and the final column is the minimum of the first two columns. The total score is the sum of the average of the three highest quiz scores, the points awarded based on the extra credit score, and bonus points for improvement. The letter grade is A, B, C, or D if the total score is greater than or equal to 90, 80, 70, or 60, respectively, and is an F if the total score is less than 60.

In our first performance experiment, all non-constant formulas in the spreadsheet were edited, thus triggering Tasks 1, 2, and 4. We term this the *user-edit experiment.* In our second performance experiment, we validated the minimum number of cells needed to achieve full coverage (Task 3). We term this the *user-validate experiment.* To compare runtimes of the two approaches, we ran both experiments on a Sun workstation and compared the timings of the Region Representative approach to those of the Straightforward approach. All timings are execution time averages of ten consecutive runs, and were taken on a Sun UltraSparc with 512 MB of RAM, with a single user, under Liquid Common Lisp 5.0.3 with Garnet [20].

Task 1: The comparisons for Task 1 (collecting static information) in the user-edit experiment are graphed in Figure 12. As the graph shows, for 10-500 students, the Region Representative approach was much faster than the Straightforward approach, and it did not slow the system down appreciably for the spreadsheet of 1 student. Further, responsiveness per edit remained reasonable for the Region Representative approach, adding less than one-half second to the spreadsheet's response time per formula edit even in the 500-student spreadsheet. The polynomial growth of savings demonstrated in this experiment by the Region Representative approach is consistent with the multi-factor advantage pointed out earlier by the analysis of this task's reasoning cost. The Region Representative's growth is not flat because of the information that must be collected to support the visual aspects, as discussed in Section 5.

Task 2: The costs of Task 2 were negligible under both approaches.

Task 3: Figure 13 displays performance comparisons for Task 3, gathered in the user-validate experiment. In the experiment, the goal was to validate all cells in the spreadsheet. Under the Region Representative approach, it was possible, by selecting appropriate test cases, to achieve 100% coverage in 10 validations. The fact that at least 10 is required for this spreadsheet can be seen from the two rightmost formulas alone: `LetterGrade` has 5 cases and `Imp` has 2 cases, each of which interacts with each `LetterGrades` case. The same inputs that exercise these were also chosen so that the other interactions were exercised at the same time.



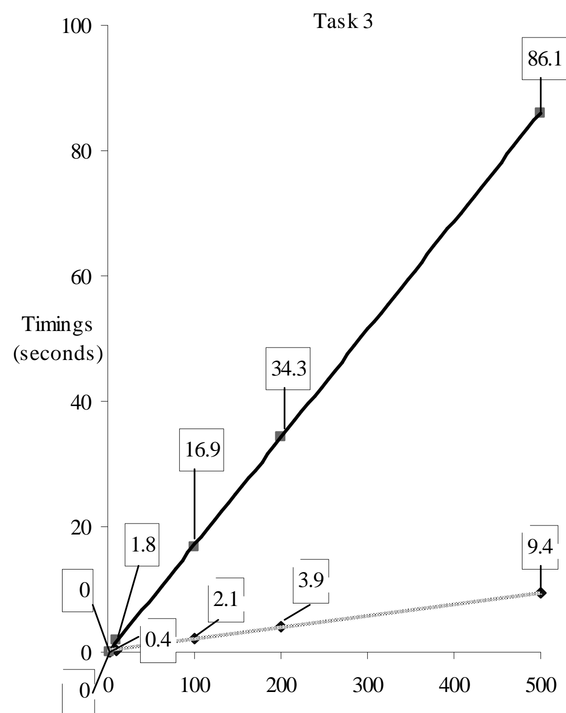


Figure 12. Task 1 costs of editing all 7 of the non-constant formulas for the different spreadsheet sizes on the x-axis. Solid black denotes the Straightforward approach, and gray/dashed denotes the Region Representative approach. Data labels marked "t" are total times incurred for these edits, and are the values actually plotted. In addition, average response times per single edit action are shown (marked "pe").

Figure 13. Total execution times of Task 3 in the user-validate experiment (validating enough cells to turn all testing borders blue or black). For both approaches, response times (not shown) were all less than one second per user validation action.

Thus, under the Region Representative approach, we achieved full coverage by validating 10 cells in the `LetterGrade` column once each, a total of 10 validations per spreadsheet, with the exception of the 1-student spreadsheet. In the 1-student spreadsheet, under both approaches, we measured timings by validating the only cell in the `LetterGrade` column once. This did not achieve full coverage in this spreadsheet, but is the most useful for consistent comparisons with the larger grids. Also, even though it does not achieve full coverage, its timing shows that the overhead expense did not swamp the costs in small grids.

On the other hand, in the Straightforward approach, each `LetterGrade` cell had to be validated 10 times to achieve full coverage, a total of 10*N* validations per spreadsheet. In addition to checking off values, in the Straightforward approach a great deal of editing of input cells was required; however, we omitted this cost, choosing instead to isolate validation cost.

Thus, the graph shows only the execution costs of checking off enough values to achieve 100% du-adequacy.

In the Task 3 measurements, the impact of the visual updating was very apparent. In the Region Representative approach, the GUI updates accounted for 50% to 90% of the total cost. In the Straightforward approach, the GUI cost was much higher than in the Region Representative approach in number of seconds (e.g., 34.9 seconds versus 9.3 seconds in the 500-student case), but the reasoning cost still outweighed the GUI cost in that approach. This is because each spreadsheet required 10 separate validation actions for each cell in the Straightforward approach, which caused 10 duplicated traversals over the same paths (greatly increasing the reasoning cost) even though only fractions of the paths required visual updating during any one traversal.

Task 4: Figure 14 presents performance comparisons of the two approaches for Task 4, collected in the user-edit experiment. Task 4 includes a visual component, which is the reason the Region Representative approach's times increased at a slow linear rate, rather than holding constant. The Region Representative approach's total costs were approximately the same as for Task 3; however, the Straightforward approach's total costs were much less expensive than for Task 3, because Task 4 features only 7 user actions (edits) per spreadsheet triggering data structure traversals, as compared to the 10N user actions (checkmarks) per spreadsheet of Task 3, which triggered many more traversals. Task 4's cost per edit (response time cost) was just over 1 second for the largest spreadsheet under the Region Representative approach, compared with almost 5 seconds for the same spreadsheet under the Straightforward approach.

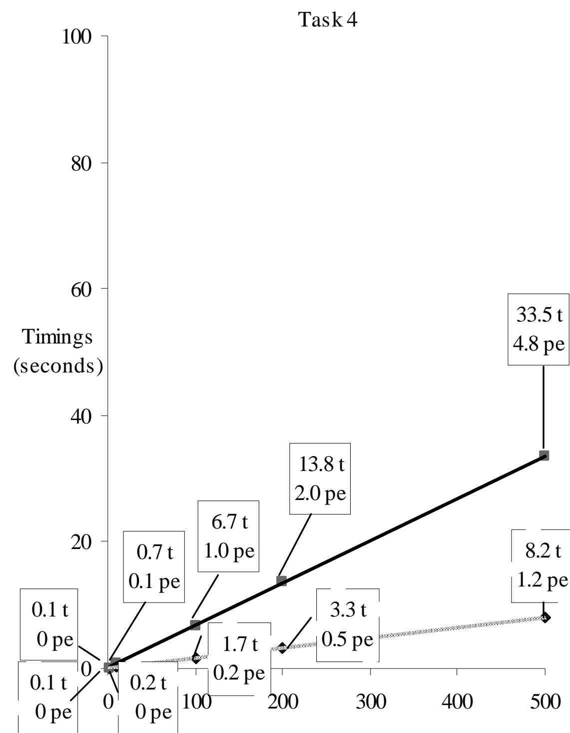


Figure 14. Task 4 comparison in the user-edit experiment.

User Actions: Table 4 shows the user actions required to perform Task 3 in the user-validate experiment, assuming that the quiz and extra credit scores had previously been entered. In the table, the user edit actions under the Region Representative approach are 0 in the best case,

which occurs if there are at least 10 students and if these students' inputs exercise all the du-associations. The probability of this being true increases with the number of students in the spreadsheet. In the worst case, the user must edit scores for 9 of the students in the grid in order to get the additional coverage needed beyond that provided by the real scores. Given the necessary input actions, 10 validations are required, one for each unique du-association path, to achieve 100% coverage. These counts follow the user-validate experiment's design, which count the minimum number of actions needed. Users could enter more inputs and validate more cells if they wished.

N-student version of `MatrixGrades` spreadsheet	Region Representative approach (actions)		Straightforward approach (actions)	
	Students whose scores must be edited	Validation actions	Students whose scores must be edited	Validation actions
1 student	9	10	9	10
10 students	0-9	10	90	10-100
100 students	0-9	10	900	10-1000
200 students	0-9	10	1800	10-2000
500 students	0-9	10	4500	10-5000

Table 4. User actions required to test the `MatrixGrades` spreadsheet (Task 3). The edit counts in the Straightforward approach are greater than the number of students because each student must be edited 9 times.

The user edit actions under the Straightforward approach come from the fact that each cell is reasoned about separately; hence to completely validate every cell, each cell must be forced through all 10 test cases. Because the user is allowed to select a set of cells and validate them with one click, the number of physical mouse clicks required to perform the *validations* could be reduced to as few as 10 if the user first rubberbands the entire group. However, there is no similar way to reduce the number of *edit* actions in the Straightforward approach: the user must enter enough inputs to force every student row through each of the 10 cases.

7. Conclusion

In previous work, we presented the WYSIWYT methodology, an approach to supporting systematic testing of individual spreadsheet cells. However, the approach was not scalable to large homogenous grids, because the costs of the methodology were highly dependent on the number of cells rather than on the number of distinct formulas. Thus, 500 cells with the same replicated formula would have to be tested individually.

An obvious approach to solving this problem is to provide what amounts to a simple user interface device, namely to rubberband large groups of cells to test them at once. In this paper

this approach is termed the Straightforward approach. In contrast, the Region Representative approach incorporates the homogeneity of spreadsheet grids into all of the theoretical model, the system's reasoning mechanisms, and the user's interactions about testedness.

Both the Straightforward and the Region Representative approaches allow a *user* validation action on one cell to be leveraged across an entire region. This reduces user actions and, in the case of the Region Representative approach, also reduces manual test case generation. However, the user action savings available under the Straightforward approach are not as great as those available under the Region Representative approach, as the experiments showed. Also, unlike the Straightforward approach, the Region Representative approach greatly reduces the *system* time required to maintain testedness data, so that it removes much of the dependency of system time on grid region size. This is critical in maintaining the high responsiveness that is expected in spreadsheet languages even in the presence of large grids. However, the time analyses and performance experiments also brought out the tension between the system completing the tasks speedily in order to maintain responsiveness in the highly interactive world of spreadsheets versus providing as much immediate visual feedback about testedness as possible, which slows down the system.

This work is part of our ongoing effort to develop an integrated, incremental approach to testing for both end users and programmers working in the spreadsheet paradigm. We have performed some empirical work [8, 25, 28], and more is planned. Our studies so far show significant testing advantages in using the WYSIWYT approach, but there are many potential pitfalls, which we are working to overcome. One possibility is that, for some spreadsheets, users may have difficulty devising suitable test cases. To address this, we are working on including automatic test case generation. We have also been working on ways to ease the user's oracle task, integration of explicit assistance for fault localization [23], and continual improvements to the visual devices that can guide users to actions that will increase testing coverage of the spreadsheet.

All the mechanisms we are incorporating into our WYSIWYT methodology are designed for tight integration into the environment, with the only visible additions being checkboxes and coloring devices. There are no testing vocabulary words such as "du-association" displayed, no dialog boxes about testing options, and no separation of testing results from the program fragments producing those results. This design reflects the goal of our research into testing methodologies for this kind of language, which is to bring at least some of the benefits that can come from the application of formal testing methodologies to spreadsheet users.

Acknowledgments

Curtis Cook and Thomas Green have made important contributions relating to the methodology's usability and usefulness to users. We also thank the other members of the Visual Programming Research Group at Oregon State University for their feedback and help with the implementation. This work was supported in part by the National Science Foundation under ESS Award CCR-9806821, Faculty Early CAREER Award CCR-9703198, and ITR Award ITR-0082265 to Oregon State University. Patent pending.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley Publishing Company, 1986.

[2] P. Brown and J. Gould, "An Experimental Study of People Creating Spreadsheets," *ACM Trans. Office Automation* 5 (3), 258-272, July 1987.

[3] M. Burnett and H. Gottfried, "Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures," *ACM Trans. Computer-Human Interaction* 5(1), 1-33, Mar. 1998.

[4] M. Burnett, A. Agrawal, and P. Zee, "Exception Handling in the Spreadsheet Paradigm," *IEEE Trans. Software Engineering* 26(10), 923-942, Oct. 2000.

[5] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang, "Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm," *J. Functional Programming*, 155-206, March 2001.

[6] M. Burnett, A. Sheretov, and G. Rothermel, "Scaling Up a 'What You See Is What You Test' Methodology to Testing Spreadsheet Grids," *Proc. 1999 IEEE Symp. Visual Languages*, 30-37, Sept. 1999.

[7] E. Chi, J. Riedl, P. Barry, and J. Konstan, "Principles for Information Visualization Spreadsheets," *IEEE Computer Graphics and Applications*, July/Aug. 1998.

[8] C. Cook, K. Rothermel, M. Burnett, T. Adams, G. Rothermel, A. Sheretov, F. Cort, and J. Reichwein, "Does a Visual 'Testedness' Methodology Aid Debugging?" Oregon State University TR #99-60-07, rev. Mar. 2001. Available at ftp://ftp.cs.orst.edu/pub/burnett/TR.EmpiricalTestingDebug.ps

[9] Coopers & Lybrand UK, "Spreadsheet Modelling in Financial Services Institutions," http://www.planningobjects.com/jungle1.htm, June 1997.

[10] R. Djang and M. Burnett, "Similarity Inheritance: A New Model of Inheritance for Spreadsheet VPLs," *Proc. 1998 IEEE Symp. Visual Languages*, 134-141, Sept. 1998.

[11] E. Duesterwald, R. Gupta, and M. L. Soffa, "Rigorous Data Flow Testing through Output Influences," *Proc. Second Irvine Software Symposium*, 131-145, Mar. 1992.

[12] P. Frankl and E. Weyuker, "An Applicable Family of Data Flow Criteria," *IEEE Trans. Software Engineering* 14(10), 1483-1498, Oct. 1988.

[13] D. Galletta, D. Abraham, M. El Louadi, W. Lekse, Y. Pollalis, and J. Sampler, "An Empirical Study of Spreadsheet Error-Finding Performance," *Accounting, Management, and Information Technology* 3(2), 79-95, 1993.

[14] D. Hamlet, B. Gifford, and B. Nikolik, "Exploring Dataflow Testing of Arrays," *Proc. Int'l. Conf. Software Engineering*, 118-129, May 1993.

[15] J. Horgan and S. London, "Data Flow Coverage and the C Language," *Proc. Fourth Symp. Testing, Analysis, and Verification*, 87-97, Oct. 1991.

[16] J. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Trans. Software Engineering* 9, 347-354, May 1993.

[17] J. Leopold and A. Ambler, "Keyboardless Visual Programming Using Voice, Handwriting, and Gesture," *Proc. 1997 IEEE Symp. Visual Languages*, 28-35, Sept. 1997.

[18] Microsoft Corporation, *Microsoft Excel 4.0 User's Guide 1 and Microsoft Excel 4.0 Function Reference*, 1992.

[19] B. Myers, "Graphical Techniques in a Spreadsheet for Specifying User Interfaces," *Proc. ACM Conf. Human Factors in Computing Systems*, 243-249, May 1991.

[20] B. Myers, D. Guise, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *Computer*, 71-85, Nov. 1990.

[21] R. Panko, "What We Know About Spreadsheet Errors," *J. End User Computing* 10(2), 15-21, Spring 1998. A longer, continuously-updated version is at http://panko.ccba.hawaii.edu/ssr/, with related information at http://panko.ccba.hawaii.edu/humanerr/ and http://panko.ccba.hawaii.edu/humanerr/SS.htm.

[22] S. Rapps, and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Engineering* 11, 367-375, Apr. 1985.

[23] J. Reichwein, G. Rothermel, and M. Burnett, "Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging," *Proc. Conf. Domain Specific Languages (DSL'99)*, 25-38, Oct. 1999.

[24] B. Ronen, R. Palley, and H. Lucas, "Spreadsheet Analysis and Design," *Communications of the ACM* 32 (1), 84-93, Jan. 1989.

[25] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov, "A Methodology for Testing Spreadsheets," *ACM Trans. Software Engineering and Methodology*, 110-147, Jan. 2001.

[26] G. Rothermel, L. Li, and M. Burnett, "Testing Strategies for Form-based Visual Programs," *Proc. Eighth Int'l. Symp. Software Reliability Engineering*, 96-107, Nov. 1997.

[27] G. Rothermel, L. Li, C. DuPuis, and M. Burnett, "What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs," *Proc. Int'l. Conf. Software Engineering*, 198-207, Apr. 1998.

[28] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, and G. Rothermel, "An Empirical Evaluation of a Methodology for Testing Spreadsheets," *Proc. Int'l. Conf. Software Engineering*, 230-239, June 2000.

[29] T. Smedley, P. Cox, and S. Byrne, "Expanding the Utility of Spreadsheets Through the Integration of Visual Programming and User Interface Objects," *ACM Proc. Workshop on Advanced Visual Interfaces*, 148-155, May 1996.

[30] T. Teo and M. Tan, "Quantitative and Qualitative Errors in Spreadsheet Development," *Proc. Thirtieth Hawaii Int'l. Conf. System Sciences,* Part 3, Vol. 3, 149-155, Jan. 1997.

[31] G. Wang and A. Ambler, "Solving Display-Based Problems," *Proc. 1996 IEEE Symp. Visual Languages*, 122-129, Sept. 3-6, 1996.

[32] N. Wilde and C. Lewis, "Spreadsheet-Based Interactive Graphics: From Prototype to Tool," *Proc. ACM Conf. Human Factors in Computing Systems*, 153-159, Apr. 1990.