2006

Scaling a Dataflow Testing Methodology to the Multiparadigm World of Commercial Spreadsheets

Marc Randall Fisher II
University of Nebraska-Lincoln, fisherii@google.com

Gregg Rothermel
University of Nebraska-Lincoln, gerother@ncsu.edu

Tyler Creelan
Oregon State University

Margaret Burnett
Oregon State University, burnett@eecs.oregonstate.edu

Scaling a Dataflow Testing Methodology to the Multiparadigm World of Commercial Spreadsheets

Marc Fisher II, Gregg Rothermel
University of Nebraska-Lincoln
{mfisher, grother}@cse.unl.edu

Tyler Creelan, Margaret Burnett
Oregon State University
{creelan, burnett}@eecs.oregonstate.edu

Abstract

Spreadsheets are widely used but often contain faults. Thus, in prior work we presented a dataflow testing methodology for use with spreadsheets, which studies have shown can be used cost-effectively by end-user programmers. To date, however, the methodology has been investigated across a limited set of spreadsheet language features. Commercial spreadsheet environments are multiparadigm languages, utilizing features not accommodated by our prior approaches. In addition, most spreadsheets contain large numbers of replicated formulas that severely limit the efficiency of dataflow testing approaches. We show how to handle these two issues with a new dataflow adequacy criterion and automated detection of areas of replicated formulas, and report results of a controlled experiment investigating the feasibility of our approach.

1. Introduction

Spreadsheets are used by a wide range of end users to perform a variety of important tasks, such as managing retirement funds, performing tax calculations, and forecasting revenues. Evidence shows, however, that spreadsheets often contain faults, and that these faults can have severe consequences. For example, spreadsheet errors caused Shurgard Inc. to overpay employees by $700,000 [21] and cost Transalta Corporation 24 million dollars through overbidding [8].

Researchers have been responding to these problems by creating approaches that address dependability issues for spreadsheets, including unit inference and checking systems [1, 2], visualization approaches [6, 20], interval analysis techniques [3, 4], and approaches for automatic generation of spreadsheets from a model [9]. Commercial spreadsheet systems such as Microsoft Excel have also incorporated several tools for assisting with spreadsheet dependability, including dataflow arrows, anomaly detection heuristics, and data validation facilities.

In our own prior research, we have presented an integrated family of approaches to help end users improve the dependability of their spreadsheets, called the "What You See is What You Test" (WYSIWYT) methodology. At the core of this methodology is a testing approach that helps spreadsheet users identify problems in interactions between cell formulas – a prevalent source of spreadsheet errors [14]. We have augmented this methodology with techniques for automated test case generation [12], fault localization [19], and test reuse and replay mechanisms [10]. Our studies of the WYSIWYT methodology itself [15, 18] suggest that it can be effective, and can be applied by end users with no specific training in the underlying testing theories.

Results such as these are encouraging; however, to date, our work on spreadsheet dependability mechanisms, and our studies of them, have been performed in the context of the research spreadsheet environment Forms/3. Commercial spreadsheet environments are multiparadigm languages with features such as higher-order functions (functional paradigm), table query constructs (database query languages), user-defined functions (implemented in an imperative sublanguage), meta-program constructs, and pointers, and these features are not accommodated by prior approaches. In addition, most spreadsheets have large areas of replicated formulas which require some form of aggregation and abstraction to allow our methodologies to scale reasonably (i.e., operate sufficiently efficiently). The only previous approach to consider testing methodologies for spreadsheet regions [5] has required a form of region declaration, and thus does not provide *unassisted* discovery of the testing needs of the informal regions that exist in commercial spreadsheets.

In this paper, we address these two problems. To support multiparadigmatic features, we devised a generalization of our prior test adequacy criterion that con-

17th International Symposium on Software Reliability Engineering (ISSRE'06)
0-7695-2684-5/06 $20.00 © 2006 IEEE

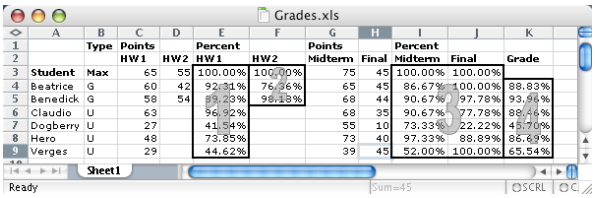


Figure 1. An Excel spreadsheet. The numbered rectangles are referenced in Section 4.

siders functions in the formulas to determine their patterns of execution. For replicated formulas, we implemented a family of techniques for combining them into regions. Throughout this work, we focus on Excel, the de-facto standard commercial spreadsheet environment, but our methodology could be extended to the wide variety of Excel work-alike environments, e.g. OpenOffice/StarOffice or Gnumeric.

To assess the resulting new methodology we performed an experiment within a prototype Excel-based WYSIWYT system on a set of non-trivial Excel spreadsheets. This experiment evaluates the costs of our methodology along several dimensions, and also compares the different techniques we have devised for finding regions to a baseline (no-regions) approach. Our results suggest that our algorithms can support the use of WYSIWYT on commercial spreadsheets; they also reveal tradeoffs among the region inference algorithms.

2. Background: WYSIWYT

The WYSIWYT methodology [4, 12, 10, 17, 19] provides several techniques and mechanisms with which end-user programmers can increase the dependability of their spreadsheets. Underlying these approaches is a dataflow test adequacy criterion that helps end users incrementally check the correctness of their spreadsheet formulas as they create or modify a spreadsheet. End-user support for this approach is provided via visual devices that are integrated into the spreadsheet environment, and let users communicate testing decisions and track the adequacy of their testing efforts.

The basic computational unit of a spreadsheet is a cell's formula. Thus, our adequacy criterion is developed at the granularity of cells. Since many of the errors in spreadsheets are reference errors, we focus on dependencies between cells. This allows us to catch a wide range of faults, including reference, operator, and logic faults.

The test adequacy criterion underlying WYSIWYT is based on a model of a spreadsheet called the Cell Relation Graph (CRG). Figure 1 shows an Excel spreadsheet, *Grades*, and Figure 2 shows a portion of the CRG corresponding to row 4 of that spreadsheet. In the CRG, nodes correspond to the cells in the spreadsheet. Within

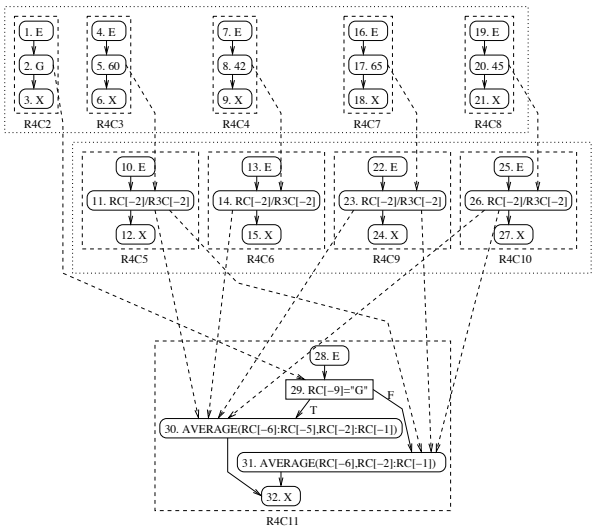


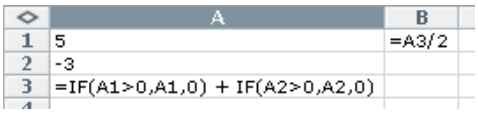
Figure 2. A CRG for the *Grades* spreadsheet

each CRG node there is a cell formula graph (CFG) that uses nodes to represent subexpressions in formulas, and edges to represent the flow of control between subexpressions. The CFG has two types of nodes, predicate nodes such as node 29 in *R*4*C*11, and computation nodes such as node 30 in *R*4*C*11.

The edges between CFGs in the CRG in Figure 2 represent *du-associations*, which link definitions of cell values to their uses. A *definition* is an assignment to a cell of a value; each computation node provides a definition of the cell in which it resides. A *use* of a cell *C* is a reference to *C* in another cell. For each use *U* of cell *C*, a du-association connects each definition of *C* to *U*. CRGs can be generated efficiently for a spreadsheet using the algorithms presented in Reference [17].

Based on the CRG model, we defined the *output influencing definition-use adequacy criterion (du-adequacy)* for spreadsheets. Under this criterion, a du-association is considered exercised if, given the current inputs, both the definition and the use node are executed, and the cell containing the use or some cell downstream in dataflow from it is explicitly marked by a user as containing a value that is valid given the current assignment of values to other cells. A test suite is considered adequate if all feasible (executable) du-associations in the CRG are exercised.

Spreadsheets often contain many duplicated formulas. In such cases it is impractical to require a tester to make separate decisions about each cell containing one of these duplicated formulas. Thus, in prior work [5], we extended WYSIWYT to handle regions of duplicate formulas. In that approach, a *region* is a set of cells explicitly identified by the user as sharing the same formula. (It is also possible that regions could be identi-



	A	B
1	5	=A3/2
2	-3	
3	=IF(A1>0,A1,0) + IF(A2>0,A2,0)	

Figure 3. Spreadsheet fragment

fied based on copy/paste actions). Figure 2 shows the regions identified by this process as boxes drawn using dotted lines around the cells included in each region. Cells *R*4*C*5, *R*4*C*6, *R*4*C*9, and *R*4*C*10 form a region and all of the input cells (cells containing constants rather than formulas) form another region.

To extend the du-adequacy criterion to spreadsheets containing such regions we grouped nodes and du-associations. Within a given region, two CFG nodes are *corresponding* if they are in the same location in their respective CFGs. In Figure 2, CFG nodes 11, 14, 23, and 26 are corresponding nodes. We defined an equivalence class relationship over du-associations such that two du-associations are in the same class if and only if their definition nodes are corresponding and their use nodes are corresponding. In Figure 2, du-associations (11, 30), (14, 30), (23, 30), and (26, 30) are in the same equivalence class. Our modified adequacy criterion stated that if any du-association in an equivalence class is tested, then all of the du-associations in that class are tested.

3. Supporting the Multiparadigmatic Nature of Cell Formulas

In Section 2, we presented the du-adequacy criterion that has been used in WYSIWYT research to date based on the CRG model of spreadsheets, but as outlined in Section 1, there are formula constructs in commercial spreadsheet languages that this du-adequacy criterion does not support. For example, consider cell *A*3 in Figure 3. With two `IF` expressions added together, it is unclear what the definitions for *A*3 are. We illustrate our new adequacy criterion by first describing how we handle this (still purely declarative) subtlety, and then demonstrate the criterion's ability to scale to multiparadigmatic aspects of spreadsheets.

We decompose the problem of handling formulas into two steps. The first step involves identifying *sources*, a generalized form of definitions that represent part of a cell's computation, and *destinations*, a generalized form of uses. The second step involves connecting sources to destinations to define interactions between cells that need to be tested. To show how this process works, we walk through it using Figure 3.

To determine the cell interactions for this example, we need to determine the sets of sources and destinations for each of the cells. Cells *A*1, *A*2 and *B*1 are

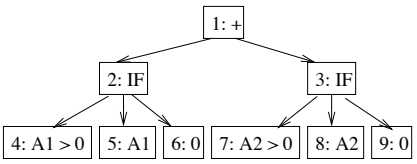


Figure 4. AST for formula in *A*3

simple cases that can be handled in the same fashion as in previous versions of WYSIWYT. Any formula that does not include conditional functions, functions that operate on or return references, or user-defined functions has only a single source. Any references in such a formula become destinations.

Cell *A*3 is more interesting. To facilitate discussion of its handling we use the AST in Figure 4. To determine sources for complex formulas such as this, we follow two steps. The first step is to identify the *source components* that represent different patterns of computations that can be performed by functions in the formula. The second is to combine these source components into the sources that represent the patterns of computation for the formula.

The formula for Cell *A*3 contains two function calls that need to be considered; namely each of the `IF` subexpressions. All `IF`'s have two possible patterns of evaluation, one that corresponds to the predicate evaluating to true, and one that corresponds to the predicate evaluating to false. We would like to capture these differing patterns of evaluation in the definition of our source components. One approach we considered was to convert all Excel functions into an equivalent UDF, and use the technique described later in Section 3.2 to determine source components and destinations. However, because this requires at least as much effort as considering the functions individually (since we do not have access to source code for the built-in functions, we would have to reverse-engineer UDF code for each of them), and because of imprecisions involved in the handling of UDFs, we chose to consider them individually. Consider the first `IF` (node 2 and its children in the AST); for this `IF`, we recognize two source components, (2, `T`) and (2, `F`). (The 2 indicates the AST node, and `T` or `F` indicates which "behavior" we are interested in). Similarly, for node 3 and its children we create the source components (3, `T`) and (3, `F`).

The source components are combined to form sources for cell *A*3. We consider two methods for doing this. One method is to consider sets of feasible combinations of source components. For cell *A*3, these combinations are {(2,`T`), (3, `T`)}, {(2, `T`), (3, `F`)}, {(2, `F`), (3, `T`)} and {(2, `F`), (3, `F`)}. For the current input assignment, the source {(2, `T`), (3, `F`)} is exercised. This method captures all of the possible computation pat-

terns for the formula and could be used when particularly rigorous testing is needed, but generates a number of sources exponential in the number of function calls in the formula.

The second method is to create a source for each source component in the formula. This creates fewer sources (in general), and on any given execution, allows multiple sources to be exercised. In our example, for the given inputs, sources (2, T) and (3, F) would be exercised. For the rest of the discussion, we assume we are using this simpler method.

Destinations for $A3$ are defined in the same way as uses were for du-adequacy. The destinations are (4, $A1$, T), (4, $A1$, F), (5, $A1$), (7, $A2$, T), (7, $A2$, F), and (8, $A2$).

Next we build a set of *interactions* that we wish to test. As we did with du-adequacy, we consider all source-destination pairs. For the example we have been considering these are {($A1$, (4, $A1$, T)), ($A1$, (4, $A1$, F)), ($A1$, (6, $A1$)), ($A2$, (7, $A2$, T)), ($A2$, (7, $A2$, F)), ($A2$, (9, $A2$), ((2, T), $B1$), ((2, F), $B1$), ((3, T), $B1$), ((3, F), $B1$)}.

Since the process of generating source components, sources, and destinations is syntax-driven, it can be automated using standard parsed representations (such as ASTs) of cell formulas. In addition, determining which source components and destinations are exercised requires only execution traces of the formulas, which are easy to gather in a spreadsheet engine [17].

An additional question involves the interaction of our new du-adequacy criterion with the region mechanism for handling duplicated formulas described in Section 2. In that description we defined corresponding definitions and uses, and used those to define corresponding du-associations. For our new du-adequacy criterion we can use a similar process, defining corresponding source components and destinations based on the locations of the constructs in the cell formulas. Then two sources, S_1 and S_2, are corresponding if for each source component C_i in S_1 there is at least one corresponding source component in S_2, and for each source component C_j in S_2 there is at least one corresponding source component in S_1. Interactions are considered corresponding if their sources and destinations are corresponding.

3.1. Handling Built-in Excel Functions

The previous section described our new adequacy criterion, but we still have to demonstrate how it can be applied to the built-in Excel functions that give rise to the multiparadigmatic nature of the language. To facilitate consideration of this, we partition the built-in functions into a small number of classes according to language features to which they relate: higher-order functions, meta-programming constructs, pointers, querying, and matrix operations. These partitions include all of the functions listed in the Excel 2003 documentation that are purely functional (Excel also includes functions such as NOW that access the state of the environment) and are not strictly computational (functions such as SUM and AVERAGE that perform simple arithmetic procedures on their parameters).

3.1.1. Handling higher-order functions. Although higher-order functions are often considered to be a programming language feature commonly associated with functional programming languages, there is support for a form of higher-order functions in Excel formulas. More precisely, Excel has a small number of functions that allow the dynamic construction of predicate expressions used for simple iterative computations, including SUMIF, COUNT, COUNTA, COUNTBLANK, and COUNTIF. To show how our approach handles these, we consider the formula $= \texttt{SUMIF}(A1 : A2, \text{“}> 0\text{”})$. The first parameter of SUMIF is a reference to a range of cells. The second parameter is a predicate to be applied to each of the cells referred to by the first parameter, which, if it evaluates to true, causes that cell's value to be added to the running total. We can convert the SUMIF into a corresponding formula using addition and IF. For our example, this would be $= \texttt{IF}(A1 > 0, A1, 0) + \texttt{IF}(A2 > 0, A2, 0)$. Notice that this transformed version is the same as the formula in cell $A3$ of Figure 3, and the source components and destinations are the same.

One issue with this method is that it generates sources and destinations for each of the IF functions, without consideration for the symmetry between the IF expressions. To address this, we can exploit the symmetry in a fashion similar to that used for regions. By defining sets of corresponding source components and destinations, and applying the modified du-adequacy criterion, we can greatly reduce the number of interactions. In the above example, (2, T) and (3, T) are one set of corresponding source components, and (4, $A1$, F) and (7, $A2$, F) are one set of corresponding destinations.

3.1.2. Handling meta-programming constructs. Excel includes a class of functions that allow meta-programming constructs. Meta-programming constructs allow programming logic based on attributes of the source code rather than attributes of the data. These include ISBLANK, CELL, AREAS, COLUMN, COLUMNS, ROW, and ROWS. ISBLANK is a predicate that returns true if and only if the referenced cell's formula is blank. CELL allows a user to query for cell formatting, protection, and address information. AREAS, COLUMNS, and ROWS return information about

the number of areas, columns, or rows included in a cell reference. COLUMN and ROW return the position (column or row) of the first cell in a cell reference. For each of these functions, the important thing to note is that they do not operate on values, and instead operate on features of the spreadsheet akin to the source code of most other languages. Consider the formula $=\texttt{ROW}(A1)$. This formula returns the value 1, regardless of the value in cell $A1$. Therefore we do not create destinations for the references in parameters to these functions or propagate testing decisions to the referenced cells.

3.1.3. Handling pointer constructs. Excel has three functions that are similar to pointer arithmetic as found in some imperative languages such as C: INDIRECT, OFFSET, and INDEX. Consider the formula $=\texttt{OFFSET}(A1,B1,C1)$. Assume that cells $B1$ and $C1$ have values 1 and 2 respectively. In this case, the call to OFFSET returns a reference to cell $C2$ (1 row down and 2 columns right from cell $A1$). There are two potential issues with these functions. First, they can use references in their arguments. For INDEX and OFFSET, the first argument is a reference to a cell or range that is used as a starting point, and the additional arguments provide an offset relative to the original cell or range. Since the value in the range referred to in the first argument ($A1$ in the example) is not used, we do not create any destinations for this reference or propagate testing decisions back to the referencing cells. However, any references used in the other arguments ($B1$ and $C1$) are dereferenced, and the corresponding values (1 and 2) are used in the calculation, therefore we can create destinations for these references and propagate testing decisions to the referenced cells just as we do for computational functions.

The second issue with these functions is the handling of the returned reference ($C2$ in the example). For purposes of *propagating* testing decisions, it makes sense to treat the returned reference as we would a regular reference. The issue of *generating* destinations for the returned reference is more complicated. In general, these functions allow a reference to any cell in any spreadsheet ever created, although in practice their use will be much more limited (for INDEX we know the returned reference will be in the range provided in the first parameter, and for OFFSET we know the returned reference will be in the worksheet referenced in the first parameter). Since in many cases it may be intractable to calculate all of the references that can be returned by these functions, we require an approximation to determine which destinations to create.

There are several approaches that can be used for this. We could create no destinations for the returned reference; this minimizes the effort required of both the system and the user testing the spreadsheet, but may cause some interactions to be untested. We could generate the set of destinations based on the history of the spreadsheet by keeping track of the returned references of these functions and creating a new destination any time a cell that had not been used before is referenced. This method forces the user to make testing decisions that are influenced by each of the interactions seen by the system, but could still miss possible interactions. It also has the undesirable effect of having input cell changes potentially change the testedness of the spreadsheet (by creating new, necessarily untested, interactions). A third possibility is to create destinations for any cells that could be referenced by the function (in the case of INDIRECT, we would limit this to cells in the workbook containing the function call). This would prevent the methodology from missing any interactions, but could create a large number of infeasible interactions. Further experimentation is needed to determine which of these possibilities is best, but for now our prototype does not create any destinations for the returned references.

3.1.4. Handling query constructs. Excel has four functions, LOOKUP, HLOOKUP, VLOOKUP, and MATCH, that search for values in a range or array and return either a corresponding value or position. These are similar to standard query operations found in database query languages. Consider the formula $=\texttt{HLOOKUP}(6,A1:B3,2)$: the function searches through the cells in the top row of the range $A1:B3$, in order from left to right, until it finds a cell with a value equal to or greater than 6, and returns a corresponding value from the second row of the range $A1:B3$.

For these functions, we use a method similar to that used for higher-order functions, converting the function to a series of nested IF expressions and defining corresponding source components and destinations. The formula $=\texttt{HLOOKUP}(6,A1:B3,2)$ is converted to $=\texttt{IF}(A1>=6,B1,\texttt{IF}(A2>=6,B2,\texttt{IF}(A3>=6,B3,\texttt{\#N/A!})))$. This formula has two sets of corresponding destinations, $\{A1,A2,A3\}$ and $\{B1,B2,B3\}$ and three sets of corresponding source components, $\{(\texttt{IF}_1,\texttt{T}),(\texttt{IF}_2,\texttt{T}),(\texttt{IF}_3,\texttt{T})\}$, $\{(\texttt{IF}_1,\texttt{F}),(\texttt{IF}_2,\texttt{F})\}$ and $\{(\texttt{IF}_3,\texttt{F})\}$.

3.1.5. Handling matrix constructs. Excel has several matrix processing functions (Excel uses the term arrays) such as MMULT. Formulas using these functions are typically assigned to a range of cells. Although there is

```
function SUMGREATERTHAN(R, V)
1. total = 0
2. for each cell C in R
3.   if C > V then
4.     total = total + C
5. return total
```

Figure 5. A user-defined function

some similarity between these ranges and the regions with shared formulas as used in Forms/3, it is primarily superficial. A matrix formula in Excel computes a single value (that happens to be a matrix), and "distributes" the value over a range of cells. In our new methodology, cells that participate in a matrix formula are treated as an aggregate cell with a single decision box to validate the value of that cell. When validated, the testing decision propagates backwards through all referenced cells. References to cells involved in a matrix formula are treated as normal destinations with a single source, but unless the reference is a range reference that includes all cells involved in the formula, testing decisions are not propagated backwards through the formula. In all other respects, matrix functions are treated as simple computational functions.

3.2. Handling Imperative Code in Spreadsheets

Excel allows imperative code to be added to spreadsheets for a variety of tasks. One of the most common uses is for creating user-defined functions (UDFs). To integrate UDFs into our new adequacy criterion, we need to statically determine the source components and destinations relevant to those UDFs, and dynamically determine which source components and destinations are exercised when tests are applied. We use program analysis techniques on the UDFs to determine the source components and destinations.

To determine the destinations in the UDF, we consider references in the parameters of the UDF. For each reference, we create a destination. To determine which destinations are executed, we use dynamic slicing on the return value of the UDF. In the case of a range being passed in as a parameter to the UDF, we create a destination for each cell in the range, and classify these destinations as corresponding destinations (similar to the corresponding destinations created for `SUMIF`). Therefore, for the formula $= \texttt{SUMGREATERTHAN}(A1 : A2, 0)$, the destinations are $\{A1, A2\}$, and they are corresponding destinations. If the functionally equivalent formula in $A3$ in Figure 3 was replaced with this formula, both destinations would be considered exercised (and would in fact be considered exercised regardless of the inputs). This difference is one of the reasons we have chosen to handle the built-in functions on a case-by-case basis rather than by converting them into equivalent UDFs.

Determining the source components of the UDF is more complicated. Since source components represent subcomputations of formulas, one approach is to consider the subcomputations, or statements (which can be generalized to flow graph nodes), of the function. Then we have a source component for each statement. For `SUMGREATERTHAN`, the source components are $\{1, 2, 3, 4, 5\}$, and if the formula considered above was substituted for the formula in cell $A3$ in Figure 3, all of these source components would be considered exercised (if $A1$ was changed to a value less than 0, then 4 would not be exercised); again this is weaker than the source components for the equivalent `IF` or `SUMIF` expressions.

4. Handling Replicated Formulas

The notion of aggregating cells into regions of similar cells in spreadsheets is not new. For example, Sajaniemi defines a number of methods for doing so [20], and others have extended his definitions [6]. However, prior work has focused on using these regions for visualization and auditing tasks. To use regions for our testing methodology we require that it be possible to define corresponding source components and destinations between the cells in the region, and to efficiently update regions as formulas change; neither of these requirements is met by the approaches of [6, 20].

We divide the task of inferring regions into two sub-tasks. The first subtask involves determining whether cells are similar, and the second involves grouping similar cells into regions.

4.1. Determining Whether Cells are Similar

The first step in developing a region inference algorithm is to define a criterion for determining whether two cells belong in the same region. Work by Sajaniemi [20] defines a number of equivalence relationships over cells. For our purposes, we consider his *formula equivalence* and *similarity* relationships, and define a variation on these that we call *formula similarity*.

Two cells are formula equivalent if and only if one cell's formula could have directly resulted from a copy action applied to the other cell's formula. Sajaniemi goes on to show that, under a certain referencing scheme, formula equivalence can be determined by textual comparison of the formulas. Most commercial spreadsheets include support for the necessary referencing scheme; in Excel it is called R1C1-style.

Sajaniemi defines two cells as being similar if and only if they are formula equivalent and *format equivalent* (two cells are format equivalent if all formatting options, e.g. font, background color, or border color, are the same), or neither contains any references to other cells and they are format equivalent. In order to find

regions in the widest variety of situations we choose to ignore format equivalence. Therefore, we define two cells as *formula similar* if and only if they are formula equivalent or neither contains any references to other cells.

4.2. Finding Regions

The second issue we considered when defining our region inference techniques is the spatial relationships between cells. Prior work has focused on rectangular areas. However, it is not necessary that regions be rectangular, and by allowing non-rectangular regions we allow larger regions to be found, thereby decreasing testing and computational effort (as well as avoiding problems with updating rectangular regions). Therefore, we consider three different candidate spatial relationships for inferring regions: discontiguous, contiguous, and rectangular. For each relationship, we describe our algorithm for finding regions in an existing spreadsheet, and we then discuss mechanisms for incrementally updating regions as the spreadsheet is updated (algorithms and run time analyses are available in Reference [13]).

4.2.1. Discontiguous regions. Using formula similarity and no additional constraints yields the most general concept of what constitutes a region: all cells in a worksheet that are formula similar are in the same region. Under this concept, regions can be discontiguous, containing cells that are not neighbors.

Discontiguous regions can be identified by iterating through the cells in a spreadsheet and looking up region identifiers in a hashtable indexed by cell formula. This process is linear in the number of cells. This technique finds two regions in *Grades* (Figure 1): (1) the cells in the areas labeled 1, 2 and 3, and (2) the cells in the area labeled 4.

To incrementally update regions there are several operations to consider. A cell's formula could be changed (through user entry or a copy/paste operation), a cell could be inserted into the spreadsheet, or a cell could be deleted from the spreadsheet. First suppose cell *C*'s formula is changed. In this case, *C* is removed from the region it is in, and if *C* is the only cell in its region, that region is deleted. Next the technique finds the region to which *C* should be added; this is done by looking up the new region in the hashtable used to find the regions initially. This is a constant time operation.

When a cell (or cells) is (are) added to a spreadsheet, all of the cells below (or to the right of, at the user's discretion) the inserted cell are shifted down (or to the right). This also causes references to the shifted cells to be updated to reflect the cells' new locations. Each cell that references a cell that is shifted must have its region information updated. References change in a similar manner when cells are deleted from the spreadsheet, and are treated similarly.

4.2.2. Contiguous regions. The discontiguous algorithm is simple and efficient; however, it is important to consider what kinds of regions end users will be able to make use of. Allowing discontiguous regions requires the creation of some device to indicate the relationship between the disconnected areas that comprise regions, which could be difficult to do in a fashion that users can understand and use. Therefore, it may be useful to require regions to be contiguous.

To find contiguous regions, our technique iterates through the cells in a spreadsheet, comparing their formulas to those of their neighboring cells, and merging formula similar cells into regions. With an efficiently implemented merge operation, the cost of this approach is linear in the number of cells in the spreadsheet. This technique finds three regions in *Grades* (Figure 1): (1) the cells in the areas labeled 1 and 2, (2) the cells in the area labeled 3, and (3) the cells in the area labeled 4.

With contiguous regions, to update regions when a formula in cell *C* in region *R* is changed, there are two factors to consider. First, *C* is removed from *R*, but then it must be determined whether *C* is required to keep two or more areas of *R* connected. This can occur only if two or more of the cells adjacent to *C* were in *R*. To determine whether *R* should be split, a search is performed on the cells in *R* starting with one of the cells adjacent to *C*. If all cells in *R* that were adjacent to *C* can be reached, it is not necessary to split the region. If any adjacent cells are not reached in the search, then the cells traversed in the search must be split off from the rest of the region. If two or more adjacent cells are not reached, the search process is repreated with another adjacent cell. In addition, it is also possible that changing the formula allows two neighbor regions to be merged. If the changed cell now has the same formula as two of its neighbors and those cells are in different regions, they need to be merged. Because of the need to potentially split or merge regions, this operation is linear in the size of *R* and of any other regions adjacent to *C*. A similar procedure is performed when a cell is deleted or inserted, taking into account changing references as in Section 4.2.1.

4.2.3. Rectangular regions. Forms/3 required regions to be rectangular, and Excel users may tend to think of their spreadsheets in rectangular blocks. Thus we also consider an algorithm that creates rectangular regions. To find rectangular regions, our technique first iterates through the cells, comparing their formulas to those of the cells directly above or below, creating all regions one cell wide of maximum height. It then iter-

ates through these regions, comparing them to the regions on either side of them, and merging the adjacent regions with formula similar formulas with the same height. Again, assuming an efficient region merge algorithm, this technique is linear in the number of cells. This technique finds four regions in the Grades spreadsheet in Figure 1, one for each of the labeled areas.

When a formula in cell *C* in region *R* is changed, the region is split into five regions. This can be done in many ways, but to be consistent with our algorithm for finding regions it proceeds as follows: one region includes all cells in *R* to the left of *C*, one region includes all cells in *R* to the right of *C*, one includes the cells in *R* directly above *C*, one includes the cells in *R* directly below *C*, and the last includes only *C* (depending on where the modified cell is located in the original region, one or more of these regions may include no cells). Each of these regions is then compared with its neighbor regions to determine whether they should be merged. The total cost of this operation depends on the number of cells in the region that is broken up and its neighboring regions.

There is one important thing to note about this approach: it does not guarantee that the regions created are the same as they would be if we re-ran the batch operation. For example, in Figure 1, if the formula of cell *I*9 was changed to match the formulas in area 3, *I*9 would be assigned to its own region. However, if this formula had been the same as the formulas in area 3 when the batch operation was performed, area 3 would have been divided into two regions (one for column *I* with *I*9 and one for column *J*). (Any update algorithm that attempted to recreate the regions that were inferred by the batch rectangular regions algorithm could potentially have wide-ranging effects on the structure of the updated regions that could be confusing to the user.) A similar procedure is performed when a cell is deleted or inserted, taking into account the issues mentioned in Section 4.2.1.

5. Assessment

Ultimately, our techniques must be empirically studied in the hands of end users, to address questions about their usability and effectiveness. Such studies, however, are expensive, and before undertaking them, it is worth first assessing the more fundamental questions of whether our techniques for handling formulas and regions scale cost-effectively to real world spreadsheets, and how our different region inference algorithms perform when applied to real spreadsheets. If such assessments prove negative, they obviate the need for human studies; if they prove positive, they provide insights into the issues and factors that should be considered in designing and conducting human studies.

More formally we consider the following research questions:

RQ1: How much does the use of WYSIWYT as extended slow down commercial spreadsheets, and how does this vary with region inference algorithms?

RQ2: How much savings in testing effort can be gained by each of the region inference algorithms?

RQ3: How do the different region inference algorithms differ in terms of the regions they identify?

To investigate these questions, we implemented a prototype in Excel using Java and VBA. The Java component performs the underlying analysis required for determining du-associations and tracking coverage, while the VBA component evaluates formulas and expressions and displays our visual devices. The prototype version used for this study provides support for most of the functions described in Section 3, treating unsupported functions as simple computational functions for purposes of testing. (It does not yet support imperative code in spreadsheets.)

5.1. Experimental procedure

As objects of analysis, we drew a sample of the spreadsheets from the EUSES Spreadsheet Corpus [11], selecting from the 1826 of those spreadsheets that contained formulas and did not use macros. The 176 selected spreadsheets ranged in size from 41 to 12,121 non-empty cells, with a mean of 1,235 non-empty cells.

Our experiment involved two independent variables: region inference algorithm and spreadsheet size.

We used all three region inference algorithms described in Section 4: `D-Regions`, `C-Regions`, and `R-Regions`. As a baseline we also used a version without region inference, `No-Regions`.

To measure spreadsheet size we used the number of non-empty cells in the spreadsheet.

We explored three dependent variables: time required for analysis on load, number of interactions in the spreadsheet, and number of regions found.

To measure time for analysis on load, we measured the time that was spent in the analysis portion of loading the spreadsheet. This measure allows us to estimate how much overhead the use of WYSIWYT requires. This measure includes the time required to infer regions and find all interactions in the spreadsheet.

To approximate the testing effort required by the different region algorithms we use the number of in-

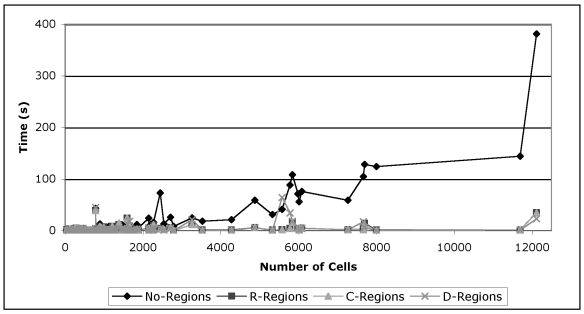


Figure 6. Analysis time on load vs. size

teractions in the spreadsheet. This works as an upper-bound on the amount of testing required, since any du-adequate test suite requires, at most, the same number of tests as there are du-associations.

Due to the properties of the algorithms, we know that if two of our region inference algorithms find the same number of regions in a spreadsheet, they have found identical regions. Thus, measuring the number of regions found lets us quickly determine whether two algorithms act identically, and we can then further inspect interesting cases when this metric differs. For the `No-Regions` algorithm, the number of regions is equal to the number of cells.

For each spreadsheet, we ran four different executions that each sequentially opened a spreadsheet, collected our measures, and then closed the spreadsheet. We did an execution for each of the four region inference algorithms utilizing the prototype Excel interface and Java analysis engine described in [7].

5.2. Data and Analysis

RQ1. Our goal with RQ1 is to determine how much our algorithms slow down the normal operation of Excel. We chose to look at load time because it is during the loading of the spreadsheet that the most work in calculating regions and interactions must occur and because previous work has demonstrated that reasonable bounds hold on the time required to respond to other user actions within the WYSIWYT methodology [17].

Figure 6 plots analysis time on load against spreadsheet size. Looking at the different techniques, it appears that the `No-Regions` approach is generally slower than the other three approaches. To explore this, we considered the differences in time between techniques using paired t-tests. As suggested by the graph, there were significant differences between `No-Regions` and the other techniques (mean differences between 9.24 and 9.69, p-values $< .05$), with no significant differences between any of the region techniques. For the techniques with regions, there does not appear to be any correlation between size of spreadsheet and time; however, with the `No-Regions` approach it appears that such a correlation might exist. A bivariate linear correlation analysis of the data resulted in a Pearson value of .863 significant with a p-value of less than .01, indicating a reasonably strong correlation between analysis time for `No-Regions` and spreadsheet size.

	No-Regions	R-Regions	C-Regions	D-Regions
interactions	1162.90	81.30	81.23	50.26
regions	1234.99	39.60	39.46	20.16

Table 1. Mean # of interactions and regions

RQ2. Table 1 shows the total number of interactions found by each technique. `No-Regions` has more than 14 times as many interactions as any of the other techniques on average (significant, paired t-test, p-value $< .05$). Both `C-Regions` and `R-Regions` had a slightly larger number of interactions than `D-Regions` (significant, paired t-test, p-value $< .05$), and approximately the same number of interactions as each other. These results suggest that testing effort could be reduced dramatically through the use of our region inference algorithms.

RQ3. Examination of the number of regions found by the different techniques shows that for 172 of the spreadsheets `R-Regions` found the same number of regions as `C-Regions`. This implies that `R-Regions` found regions identical to those found by `C-Regions` in these cases.

`D-Regions` found fewer regions than `R-Regions` and fewer than `C-Regions`, as indicated in Table 1 (significant, paired t-test, p-value $< .05$). Further examination shows that `D-Regions` found the same set of regions as `C-Regions` on only 36 spreadsheets.

5.3. Discussion

Our analysis timings show that it is feasible to perform WYSIWYT analysis on real spreadsheets, and that with region inference and our formula extensions, WYSIWYT seems to scale quite well to larger spreadsheets. In addition, from the point of view of timing, it does not seem to make much difference which region inference algorithm is used.

As expected, `D-Regions` found significantly fewer (therefore larger) regions than the other techniques, which led to fewer interactions in the spreadsheet, implying less testing effort. The lack of difference between `C-Regions` and `R-Regions`, however, was somewhat surprising, although useful. As mentioned in Section 4, `R-Regions` are difficult to update and efficient updating algorithms could lead to an inconsistent state, a problem that `C-Regions` does not suffer from. Since the vast majority of con-

tiguous regions in the study are inherently rectangular in nature, there seems to be little reason to use `R-Regions`. However, since there is a significant difference between the regions identified by `D-Regions` and `R-Regions`, user studies are needed to determine which of these methodologies provides the best balance between usability and efficiency for users.

6. Conclusions

In this paper we have presented a new test adequacy criterion, aimed at supporting not only the usual dataflow relationships between formulas, but also the more challenging multiparadigmatic features of commercial spreadsheets. We show how the adequacy criterion can be applied to Excel's support for higher-order functions, meta-programming constructs, pointer constructs, query language mechanisms, matrix constructs and user-defined functions. We also present algorithms to support the high degree of formula replication common in commercial spreadsheets. Finally, we report on the first studies of WYSIWYT to ever be conducted within a commercial spreadsheet environment.

In our continuing work, we are considering approaches for handling other features of commercial spreadsheets. Charts could be handled as a special form of cell that have targets for each cell whose value is used to generate the chart. External data sources are a form of input cell into the system; replacing them with temporary user-settable input cells would allow the user to test the logic of the spreadsheet. Using an anomaly detection mechanism on the data feeds themselves similar to that proposed in Reference [16] could help to ensure that the data feeds are reliable.

Through this work we hope to provide a system that can be used to further evaluate dependability devices with end users using large-scale spreadsheets, and in particular, that can be used in long-term studies.

Acknowledgements. This work was supported in part by the EUSES Consortium via NSF Grant ITR-0325273.

References

[1] R. Abraham and M. Erwig. Header and unit inference for spreadsheets through spatial analyses. In *Symp. on Vis. Lang. and Human-Centric Comp.*, 2004.

[2] T. Antoniu, P. A.Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In *Int'l Conf. on Soft. Eng.*, 2004.

[3] Y. Ayalew and R. Mittermeir. Interval-based testing for spreadsheets. In *Int'l Arab Conf. on Inf. Tech.*, 2002.

[4] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Int'l Conf. on Soft. Eng.*, 2003.

[5] 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. on Soft. Eng.*, 28(6):576–594, 2002.

[6] M. Clermont. Analyzing large spreadsheet programs. In *Working Conf. on Reverse Eng.*, 2003.

[7] T. Creelan and M. Fisher II. Scaling up an end-user dependability framework for spreadsheets. Technical Report 04-60-09, Oregon State University, 2004.

[8] D. Cullen. Excel snafu costs firm $24m. The Register, June 19 2003.

[9] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *Int'l Conf. on Soft. Eng.*, 2005.

[10] M. Fisher II, D. Jin, G. Rothermel, and M. Burnett. Test reuse in the spreadsheet paradigm. In *Int'l Symp. on Soft. Rel. Eng.*, 2002.

[11] M. Fisher II and G. Rothermel. The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Work. on End-user Soft. Eng.*, 2005.

[12] M. Fisher II, G. Rothermel, D. Brown, M. Cao, C. Cook, and M. Burnett. Integrating automated test generation into the WYSIWYT spreadsheet testing methdology. *ACM Trans. on Soft. Eng. and Meth.*, 15(2), 2006.

[13] M. Fisher II, G. Rothermel, T. Creelan, and M. Burnett. Scaling a dataflow testing methodology to the multiparadigm world of commercial spreadsheets. Technical Report TR-UNL-CSE-2005-0003, University of Nebraska - Lincoln, 2005.

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

[15] S. Prabhakararao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and behaviors of end-user programmers with interactive fault localization. In *Symp. on Human-Centric Lang. and Env.*, 2003.

[16] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *Int'l Conf. on Soft. Eng.*, 2002.

[17] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Tran. on Soft. Eng. and Meth.*, 27(1):110–147, 2001.

[18] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, and G. Rothermel. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In *Int'l Conf. on Soft. Eng.*, 2000.

[19] J. Ruthruff, M. Burnett, and G. Rothermel. An empirical study of fault localization for end-user programmers. In *Int'l Conf. on Soft. Eng.*, 2005.

[20] J. Sajaniemi. Modeling spreadsheet audit: A rigorous approach to automatic visualization. *J. of Vis. Lang. and Comp.*, 11(1):49–82, 2000.

[21] A. Scott. Shurgard stock dives after auditor quits over company's accounting. The Seattle Times, Nov 18 2003.