A Methodology for Testing Spreadsheets

GREGG ROTHERMEL, MARGARET BURNETT, LIXIN LI, CHRISTOPHER DUPUIS, and ANDREI SHERETOV

Oregon State University

Spreadsheet languages, which include commercial spreadsheets and various research systems, have had a substantial impact on end-user computing. Research shows, however, that spreadsheets often contain faults; thus, we would like to provide at least some of the benefits of formal testing methodologies to the creators of spreadsheets. This article presents a testing methodology that adapts data flow adequacy criteria and coverage monitoring to the task of testing spreadsheets. To accommodate the evaluation model used with spreadsheets, and the interactive process by which they are created, our methodology is incremental. To accommodate the users of spreadsheet languages, we provide an interface to our methodology that does not require an understanding of testing theory. We have implemented our testing methodology in the context of the Forms/3 visual spreadsheet language. We report on the methodology, its time and space costs, and the mapping from the testing strategy to the user interface. In an empirical study, we found that test suites created according to our methodology detected, on average, 81% of the faults in a set of faulty spreadsheets, significantly outperforming randomly generated test suites.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.6 [**Software Engineering**]: Programming Environments; H.4.1 [**Information Systems Applications**]: Office Automation; D.1.7 [**Programming Techniques**]: Visual Programming

General Terms: Algorithms, Languages, Verification

Additional Key Words and Phrases: Software testing, spreadsheets

1. INTRODUCTION

Spreadsheet languages, which are also known as form-based languages in some of the research literature, provide a declarative approach to programming, characterized by a dependence-driven, direct-manipulation working model [Ambler et al. 1992]. Users of spreadsheet languages create cells, and define formulas for those cells. These formulas reference values contained

This article is a revised and expanded version of a paper presented at the 20th International Conference on Software Engineering [Rothermel et al. 1998].

Authors' address: Computer Science Department, Oregon State University, 303 Dearborn Hall, Corvallis, OR 97331; email: grother@cs.orst.edu; burnett@cs.orst.edu.

in other cells and use them in calculations. When a cell's formula is defined, the underlying evaluation engine calculates the cell's value and those of other affected cells (at least those that are visible to the user), and displays new results.

Spreadsheet languages include, as a subclass, commercial spreadsheet systems. These systems are widely used by end-users, for a variety of computational tasks. The spreadsheet paradigm is also a subject of ongoing research. For example, there is research into using spreadsheet languages for matrix manipulation problems [Viehstaedt and Ambler 1992], for providing steerable simulation environments for scientists [Burnett et al. 1994], for high-quality visualizations of complex data [Chi et al. 1997], and for specifying full-featured GUIs [Myers 1991].

Despite the end-user appeal of spreadsheet languages and the perceived simplicity of the paradigm, research shows that spreadsheets often contain faults. For example, in an early spreadsheet study, 44% of "finished" spreadsheets still had errors [Brown and Gould 1987]. A more recent survey of other such studies reported errors in 38% to 77% of spreadsheets at a similar stage [Panko and Halverson 1996]. Of perhaps even greater concern, this survey also covers studies of field audits of "production" spreadsheets, those actually in use for day-to-day decision-making, and these studies all reported errors in at least 10% of the spreadsheets audited. Although the exact definition of "error" varied among these studies, the emphasis in most cases was on formulas that produced incorrect values. A possible factor in this problem is the unwarranted confidence creators of spreadsheets seem to have in the reliability of those spreadsheets [Wilcox et al. 1997].

In spite of this evidence, we find no discussion in the research literature of techniques for testing spreadsheets. In fact, there has been only a little work on testing in other paradigms that follow declarative models. In the domain of functional and dataflow programming, there has been work on specification-based testing (e.g., Kuhn and Frank [1997] and Ouabdesselam and Parissis [1995]), but creators of spreadsheets rarely employ formal specifications. There has also been some recent research [Azem et al. 1993; Belli and Jack 1995; Luo et al. 1992] that considers problems of testing and reliability determination for logic programs written in Prolog. However, although the logic paradigm is like the spreadsheet paradigm in that both are declarative, several features of the logic paradigm, such as the bidirectional nature of unification and backtracking after failure, are so different from the spreadsheet paradigm that the testing techniques developed for Prolog cannot be applied to the spreadsheet paradigm.

On the other hand, there has been extensive research on testing imperative programs (e.g., Clarke et al. [1989], Duesterwald et al. [1992], Frankl and Weiss [1993], Frankl and Weyuker [1988], Harrold and Soffa [1988], Hutchins et al. [1994], Laski and Korel [1993], Ntafos [1984], Offutt et al. [1996], Perry and Kaiser [1990], Rapps and Weyuker [1985], Rothermel and Harrold [1997], Weyuker [1986; 1993], and Wong et al. [1995]), and it is in this body of work that the methodology presented in this article has its

roots. However, significant differences exist between the spreadsheet and imperative programming paradigms, and these differences have ramifications for testing methodologies. These differences can be divided into four classes. The first class pertains to evaluation order. Evaluation of spreadsheets is driven by data dependencies between cells, and spreadsheets contain explicit control flow only within cell formulas. The dependence-driven evaluation model allows evaluation engines flexibility in the scheduling algorithms and optimization devices they might employ to perform computations. One common such optimization device is value caching. A methodology for testing spreadsheets must be compatible with this flexibility, and not rely upon any particular evaluation order.

The second class of differences pertains to language characteristics' impact on how precisely and how efficiently source code analysis can be done. Techniques for statically analyzing code, such as dataflow analysis and slicing, can be useful for testing (e.g., Duesterwald et al. [1992], Gupta et al. [1996], and Harrold and Soffa [1988]), but algorithms for performing these analyses on imperative programs are complicated by the presence of dynamically determined addressing and aliasing [Landi 1992; Landi and Ryder 1991]. Spreadsheets may index into arrays (grids) in formula references; however, for most spreadsheet languages, such references can be resolved at formula entry time.[1] Spreadsheets may have aliases to the extent that multiple names may refer to a single cell; however, for most spreadsheet languages these aliases, too, can be resolved statically. Furthermore, spreadsheet formulas do not contain loops, or definitions that "kill" other definitions, factors that add to the expense of analyses of imperative programs. A methodology for testing can take advantage of these simplifying factors, and obtain more efficient and precise analyses than are possible for imperative programs.

The third class of differences pertains to interactivity: spreadsheet systems are characterized by incremental visual feedback that is intertwined with the program construction process. The most widely seen example of this is the "automatic recalculation" feature. This incremental visual feedback invites the use of testing methodologies that support an incremental input and validation process. For example, when a user changes a formula, the testing subsystem should provide feedback about how this affects the "testedness" of each visible portion of the program. This raises the issue of dealing with evolving spreadsheets while maintaining suitable response time.

The fourth class of differences pertains to users of spreadsheet languages. Imperative languages are most commonly used by professional programmers who are in the business of producing software. These pro-

[1]Even table lookup formulas can be analyzed at formula entry time [Burnett et al. 1999]. This can be done by reasoning conservatively about entire groups of cells in tables, which avoids having to determine the exact table entry to which a formula refers. The approach is facilitated in spreadsheets due to the ease of grouping cells based on which formulas are the same (replicated or shared) among the table's cells.

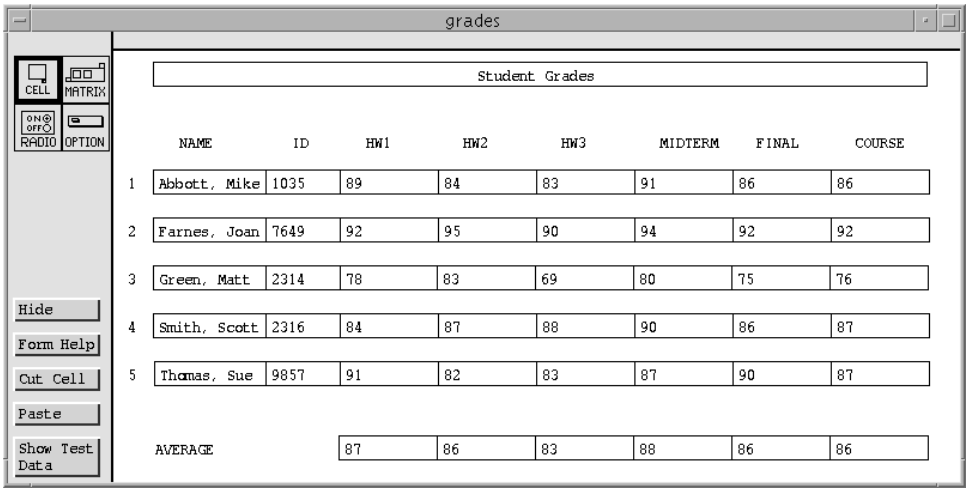

Fig. 1. Spreadsheet for calculating student grades.

grammers can be expected to know something about testing, and to place a high priority on doing a reasonably good job of testing. On the other hand, spreadsheet systems are used by a variety of users, many of whom are not professional programmers and have no interest in learning formal testing methodologies. Our goal is to provide at least some of the benefits of formal testing methodologies to these users.

This article presents a methodology for testing spreadsheets. The methodology takes advantage of the factors just described to promote efficient, precise analyses of spreadsheets, adapting data flow adequacy criteria, coverage monitoring, and dependence-based impact analysis to the task of testing spreadsheets. To accommodate the evaluation models used with spreadsheets and the interactive process by which they are created, our methodology is incremental, performing analysis and collecting the data required for testing whenever a formula is edited. To accommodate the user base of these languages, we provide an interface to the methodology that does not require an understanding of testing theory. This is accomplished through a fine-grained integration with the spreadsheet environment to provide testing information visually.

2. BACKGROUND AND DEFINITIONS

2.1 Spreadsheet Languages

Users of spreadsheet languages "program" by specifying the contents of a spreadsheet. The contents of a spreadsheet are a collection of cells; each cell's value is defined by that cell's formula. As the user enters a formula it is evaluated, and the result is then displayed. The best-known examples of spreadsheet languages are found in commercial spreadsheet systems, but there are also many research systems (e.g., Burnett and Gottfried [1998],

```
formula ::= BLANK | expr
expr ::= CONSTANT | CELLREF | ERROR | infixExpr |
         prefixExpr | ifExpr | composeExpr
infixExpr ::= subExpr infixOperator subExpr
prefixExpr ::= unaryPrefixOperator subExpr |
               binaryPrefixOperator subExpr subExpr
ifExpr ::= IF subExpr THEN subExpr ELSE subExpr |
           IF subExpr THEN subExpr
composeExpr ::= COMPOSE subExpr withclause
subExpr ::= CONSTANT | CELLREF | (expr)
infixOperator ::= +| −|∗|/| MOD | AND | OR | = | < | <=
unaryPrefixOperator ::= NOT | ERROR? | CIRCLE | ROUND
binaryPrefixOperator ::= LINE | BOX
withclause ::= WITH subExpr AT (subExpr subExpr) |
               WITH subExpr AT (subExpr subExpr) withclause
```

Fig. 2. Grammar for formulas in this article. A cell with no formula is equivalent to a cell with formula *BLANK*; the result of evaluating such a formula is a distinguished value that we term NOVALUE. The "else"-less version of an ifExpr (e.g., IF A=B THEN "A and B are the same") is simply a syntactic shortcut for the same formula with "ELSE *BLANK*" appended (e.g., IF A=B THEN "A and B are the same" ELSE *BLANK*). Parsing ambiguities do not arise in this simple language because multitoken subexpressions are always wrapped in parentheses.

Chi et al. [1997], Leopold and Ambler [1997], Myers [1991], Smedley et al. [1996], and Viehstaedt and Ambler [1992]) based on this paradigm.

In this article, we present examples of spreadsheets in the research language Forms/3 [Burnett and Gottfried 1998]. Figure 1 shows a traditional-style spreadsheet used to calculate student grades in Forms/3. The spreadsheet lists several students and several assignments performed by those students. The last row in the spreadsheet calculates average scores for each assignment; the rightmost column calculates weighted averages for each student; and the lower-right cell gives the overall course average (formulas not shown).

Figures 3 and 4 show how a user could construct a graphical clock in Forms/3. Figure 3 shows each cell with its formula. `Clock` consists of 13 cells, including two input cells (upper left) that could eventually be replaced with references to the system clock, one output cell (middle left), and several cells used in intermediate calculations (right). (We use the term *input cell* to refer to cells whose formulas contain only constants.) After the programming is finished, the cells that calculate intermediate results can be hidden, and other cells rearranged, to reach the user view shown in Figure 4.

In this article, we consider a "pure" spreadsheet language model, which includes ordinary spreadsheet-like formulas such as those described by the grammar given in Figure 2, but excludes advanced programmer-oriented features such as macros, imperative sublanguages, indirect addressing, and recursion. The grammar shown in Figure 2 reflects a subset of Forms/3, a Turing-complete spreadsheet language following this model [DuPuis and Burnett 1997]. The subset shown uses ordinary spreadsheet formulas for both numeric and graphical computations; the figures presented in this article were programmed using this subset. From this grammar, it is clear

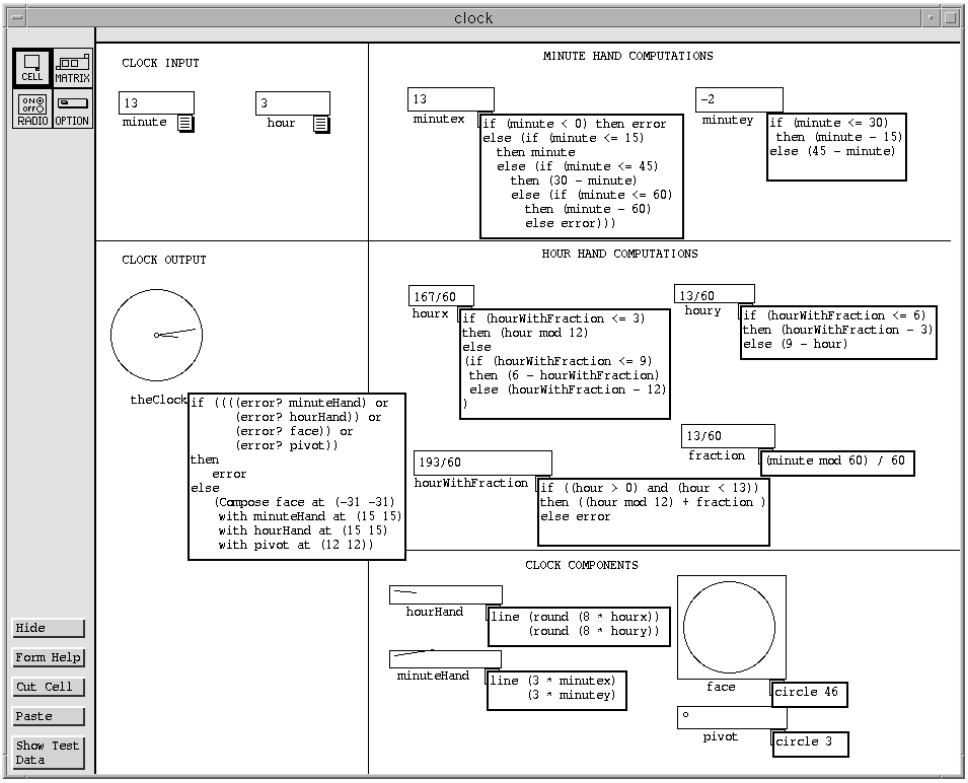

Fig. 3. Programming a clock in Forms/3.

that the only dependencies between one cell and another are data dependencies. Because of this fact, cells can be scheduled for evaluation in any order that preserves these dependencies.

2.2 Evaluation Strategies for Spreadsheet Languages

The evaluation strategies used in spreadsheet languages have a great deal of latitude regarding execution sequence, provided that all dependencies are preserved. Thus, the evaluation order of the cells in a spreadsheet depends on the data flow between source and sink cells and on whether the evaluator is using an eager or a lazy evaluation strategy, and a variety of optimizations and variations are possible. Eager evaluation is driven by changes: whenever a value of cell X is changed, the change is propagated to every cell that is affected by the change. For example, if a user edits cell X's formula, then if cell Y references X in its formula then Y is also recomputed, which in turn causes cells that refer to Y to be recomputed, and so on. Determining which cells are affected is usually done conservatively, i.e., from a static perspective.

In contrast to this strategy, lazy evaluation is driven by output: the first time a cell X is displayed, it is computed, and so is every cell that X needs. For example, if cell X is moved onto the screen through window manipulations,

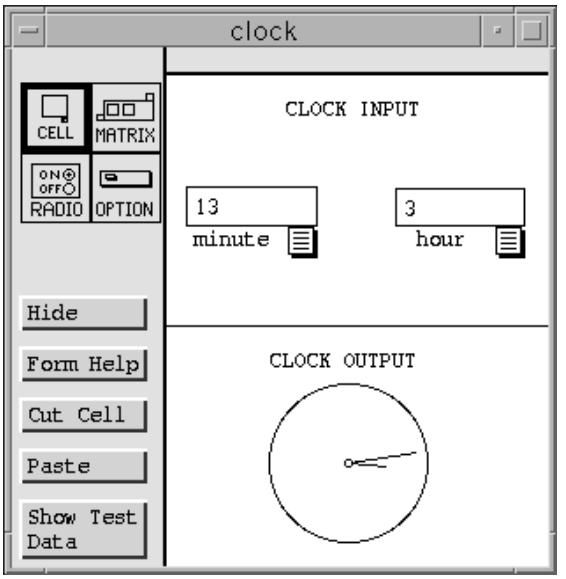

Fig. 4. The user view of the clock in Forms/3. On the input cells, formula tabs have been left visible to encourage inputting new hour and minute values. The formula tab has been hidden on the output cell.

every cell that it needs is computed (and every cell that they need, and so on) in order to finally calculate X. Whether X "needs" Y is usually determined dynamically. For example, if X's formula is "TRUE or Y," then the reference to Y will not be needed if the evaluation engine evaluates the first operand before the second. It has been shown that lazy evaluation produces the same answers as eager evaluation, provided that both terminate. However, lazy evaluation computes fewer cells.

Because spreadsheet languages tend to be visual and to display the current value of many of the cells, these values are often cached. This means that an evaluation engine also needs to keep track of which cached values are up-to-date if the user has started changing formulas. There are several methods for doing so (which are surveyed in Burnett et al. [1998]), but their mechanism is not relevant to the issues in this article.

In spreadsheet languages, some cells will be on the screen, while some will not. There are both static and dynamic mechanisms for determining which are on the screen. For example, in some languages it is possible to statically "hide" cells; in most languages the user can also scroll or otherwise move cells on and off the screen at runtime through direct manipulation. Which cells are on-screen can influence the user's testing behavior, because it determines which input cells a user can notice and attend to, and which output cells the user can see. In the case of languages following lazy evaluation, which cells are on-screen also determines which cells will be computed, since lazy evaluation is output-driven.

2.3 A Model for Spreadsheets

Test adequacy criteria provide help in selecting test data and in deciding whether a program has been tested "enough." Test adequacy criteria are often defined on models of programs rather than on code itself. We have created such a model for spreadsheet languages [Rothermel et al. 1997]; we call our model a *cell relation graph* (CRG). A CRG is a pair (V, E), where V is a set of *formula graphs*, and E is a set of directed edges connecting pairs of elements in V. Figure 5 depicts the CRG for `Clock`.[2]

Each formula graph in V models flow of control within a cell's formula, and is comparable to a control flow graph representing a procedure in an imperative program [Aho et al. 1986; Rapps and Weyuker 1985]. There is one formula graph for each cell in the spreadsheet. The process of translating an abstract syntax tree representation of an expression into its control flow graph representation is well known [Aho et al. 1986]; a similar translation applied to the abstract syntax tree for each formula in a spreadsheet yields that formula's formula graph.[3] For example, Figure 5 shows the formula graphs for the cells in `Clock`, delimited by dotted rectangles. In these graphs, nodes labeled "E" and "X" are *entry* and *exit* nodes, respectively, and represent initiation and termination of evaluation of formulas. Nodes with multiple out-edges (represented as rectangles) are *predicate* nodes. Other nodes are *computation* nodes. Edges within formula graphs represent flow of control between expressions, and edge labels indicate the value to which conditional expressions must evaluate for particular branches to be taken. In Figure 5, we have added numeric labels to the nodes to facilitate subsequent references.

The set E of edges in the CRG consists of *cell dependence edges*, which model data dependencies between cells. Figure 5 depicts these edges by dashed lines. Each edge encodes the fact that the destination cell refers to the source cell in its formula; thus, the arrows show direction of dataflow. Note that cell dependence information is typically available to evaluation engines within spreadsheet systems as a consequence of the need to evaluate formulas; thus, this information need not be specially calculated in order to construct CRGs.

Cell dependence edges do not represent control flow; this distinguishes the CRG from the control flow graphs and interprocedural control flow graphs [Landi and Ryder 1992] used to represent procedures and programs, respectively, in imperative programs. In a control flow graph or interprocedural control flow graph, an edge encodes information on execution order: the fact that a node n_2 is an immediate successor of a node n_1 in such a graph implies that execution of n_2 immediately follows execution of n_1. In a

[2]In addition to supporting test adequacy criteria, the CRG has also been used to support the application of slicing-and-dicing algorithms to spreadsheets, for use in fault localization and debugging [Reichwein et al. 1999].

[3]One language feature that merits special treatment in spreadsheet languages is the conditional subexpression. Appendix A presents details on construction of CRGs for this language feature.

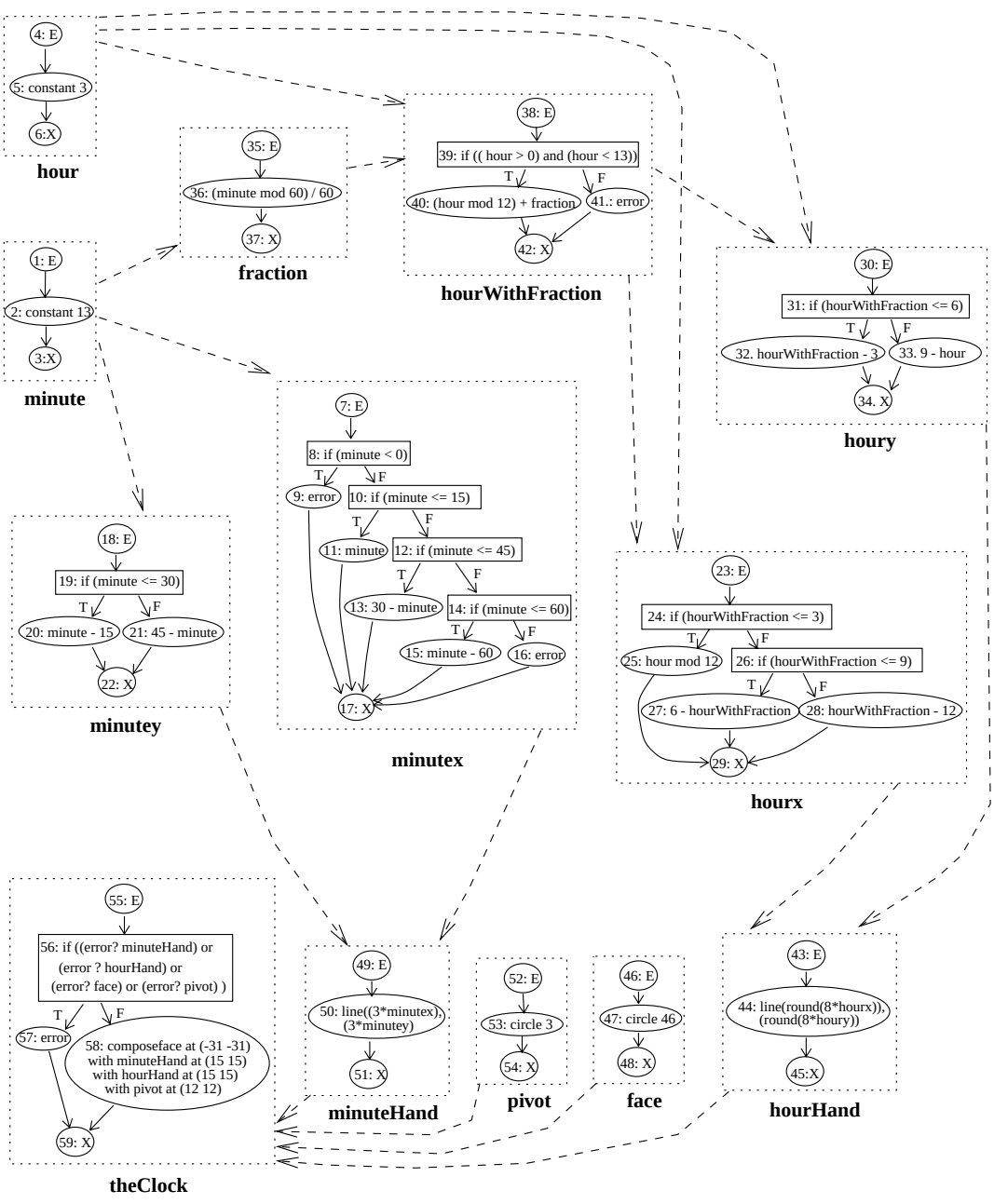

Fig. 5. Cell relation graph for `Clock`. Dotted rectangles enclose formula graphs. Solid edges depict control flow within formulas, and dashed edges depict data dependencies between cells.

cell dependence graph, the fact that formula graph F_2 is an immediate successor (along a cell dependence edge) of formula graph F_1 does *not* imply that execution of F_2 *immediately* follows execution of F_1; evaluation

engines might execute various other cells subsequent to F_1 and prior to F_2, as long as the execution preserves cell dependencies.[4]

In fact, for evaluation engines following a lazy strategy, it is not possible to determine control flow between cells statically. This is because the user's runtime navigations such as scrolling and rearranging windows are determiners of which cells must be executed, since visibility on the screen is what initiates execution. Hence, a static control flow graph of cells could not model a spreadsheet's execution under lazy evaluation.

Finally, we require a way to associate execution of formulas with CRG components. Let F be a formula with formula graph $\bar{F}$, and let F_e and F_x be the entry and exit nodes, respectively, of $\bar{F}$. An evaluation of F traverses a path through $\bar{F}$, beginning at F_e and ending at F_x. We call this path the *execution trace* for that evaluation. The existence of this execution trace is a consequence of (and attests to) the fact that, unlike the CRG as a whole, a formula graph *does* represent control flow, but only *within a single formula*. As the definitions of formula graph nodes in the next section will make clear, the granularity of the formula graph is sufficiently coarse that the formula graph can correctly model control flow within a single formula, regardless of whether evaluation is lazy or eager.

2.4 A Test Adequacy Criterion for Spreadsheets

Test adequacy criteria have been well researched for imperative languages, and various criteria have been proposed (e.g., see Clarke et al. [1989], Frankl and Weyuker [1988], Laski and Korel [1993], Ntafos [1984], Perry and Kaiser [1990], and Rapps and Weyuker [1985]). In Rothermel et al. [1997], we explored the application of several of these criteria to spreadsheets. We argued that dataflow adequacy criteria (e.g., Duesterwald et al. [1992], Laski and Korel [1993], Ntafos [1984], and Rapps and Weyuker [1985]), which relate test adequacy to interactions between definitions and uses of variables in the source code (definition-use associations), can be particularly appropriate for spreadsheets.

There are several reasons for this appropriateness. The first reason involves the types of faults that occur in spreadsheets, the largest percentage of which have been observed to involve errors in cell references [Panko and Halverson 1996]. Data flow adequacy criteria directly address this class of faults, whereas criteria analogous to the statement or decision coverage criteria commonly used for imperative programs only indirectly

[4]Another graph that might be compared to the CRG is the program dependence graph (PDG) [Ferrante et al. 1987]. However, there is little similarity between PDGs and CRGs. The CRG uses control flow edges within formula graphs—these edges are not typically present in the PDG at all. The CRG does *not* include control dependence edges, which are of central importance in the PDG. Both graphs indeed use data dependence edges, but not at the same level of granularity: in the PDG the data dependence edges are between nodes that represent statements or basic blocks; in the CRG the data dependence edges are between cells. If the CRG represented data dependence edges between the individual nodes in formula graphs, then those edges would resemble those used in the PDG.

test cell references. Second, it is easy to show, that for spreadsheets, as well as for imperative programs, dataflow adequacy criteria can produce test suites that are stronger than (subsume) criteria analogous to statement or decision coverage criteria. A third advantage of dataflow criteria involves their (relative) ease of application to spreadsheets. As mentioned in Section 1, the characteristics of spreadsheet languages allow algorithms for performing data flow analysis or slicing of spreadsheets to be more efficient and more precise than their counterparts for imperative languages (a fact that shall become more clear in the next section). Thus, the effectiveness benefits of dataflow testing can be realized, for spreadsheets, with less cost than for imperative programs, removing an obstacle that could render it less preferable to criteria based on statement or decision coverage.

To define a dataflow test adequacy criterion for spreadsheets, one approach would be to adapt the "all-uses" dataflow test adequacy criterion defined for imperative programs [Rapps and Weyuker 1985]. The all-uses criterion requires that test inputs be found that cause each executable definition-use association in the program to be exercised ("covered") by at least one path. Applied to spreadsheets, however, this criterion has drawbacks. Since end-users are not trained software engineers, it is unreasonable to assume that they will notice or pass judgment on every output that appears, even during a test. For example, in testing the formula for a student's course grade, the user may not attend to whether the overall average of all students in the class is correct. Further, in a spreadsheet, cells may be hidden or off the screen. Depending on the evaluation engine employed and the dependencies among cells, it is possible that a set of test inputs could be applied to input cells, causing definitions to reach uses that occur in hidden or off-screen cells, even though the computation involving these definitions and uses may not affect any visible output cells. Under the all-uses criterion, however, any definition-use associations exercised by a set of test inputs are considered validated, whether or not they have been observed to participate in the production of a valid output.

Instead, following the notion of the "output-influencing-All-du" adequacy criterion of Duesterwald et al. [1992], we define a data flow test adequacy criterion for spreadsheets in terms of definition-use associations that influence cell outputs. To define this criterion precisely for spreadsheets, however, we must modify several of the basic definitions of testing terminology used in the imperative programming context. In the remainder of this section we provide precise definitions, while explaining the differences between these and previous definitions, and the factors in spreadsheet testing that motivate these different definitions.

First, we define a *test case* for spreadsheet S to be a tuple (I, C), where I is a vector of input values (constant formulas) whose elements correspond to the input cells in S, and C is a cell whose value the user has examined for correctness under that input configuration. In this context, a *test* (the application of a test case by the user) is an explicit decision by the user that C's value is correct or incorrect, given the current configuration I of input

cell values. The user's act of communicating this correctness is a *validation*. After the user performs a validation act on C, C is said to be *validated*.

These definitions do *not* imply that to perform a test (to apply a test case), the user must first enter values into *all* input cells. Rather, the application of a test case involves entering values into *zero or more* input cells, while retaining the values currently contained in other input cells, and then validating (or judging incorrect) a cell value. One reason these definitions differ from those used in the imperative world is that they must formally account for the fact that spreadsheet users may not attend to any or all of the affected outputs; hence the notion of a test must include an explicit decision by the user on any one of the affected cells. A second reason is that the user's act of communicating a decision is the most important trigger for invoking the algorithms that compute coverage. A third reason these definitions differ is that they facilitate incremental testing, allowing a test to involve entering values into only a small subset of the potentially enormous set of input cells in a spreadsheet.

Next, we define definition-use associations for spreadsheets. In spreadsheets, cells serve as variables, and the value for cell C can be defined only by expressions in C's formula. Let C be a cell in spreadsheet S, with formula F and formula graph $\bar{F}$. Because formula graphs that do not contain predicate nodes always consist of exactly three nodes, if C is an input cell, then $\bar{F}$ contains only one node other than entry and exit nodes, and that node is a *definition* of C. If C is not an input cell, then each computation node in $\bar{F}$ that represents an expression referring to cell D is a *c-use* (computation use) of D and a *definition* of C. Each edge in $\bar{F}$ that has as its source a predicate node n such that n represents a conditional expression referring to another cell D is a *p-use* (predicate use) of D.

Let S be a spreadsheet with CRG G. A *definition-use association* (du-association) links a definition in G with a use in which that definition may reach. Two types are of interest. A *definition-c-use association* is a triple (n_1, n_2, C), where C is a cell, n_1 is a definition of C, and n_2 is a c-use of C. A *definition-p-use association* is a triple $(n_1, (n_2, n_3), C)$, where C is a cell, n_1 is a definition of C, and (n_2, n_3) is a p-use of C.

There are three points to consider about these definitions. First, in our definition of du-associations, the distinction between p-uses and c-uses lets us track whether a test suite that exercises all du-associations in a spreadsheet also exercises both outcomes of each predicate that contains a cell reference. For example, if a cell has a formula such as "`if x>100 then 100 else x`," treating the edges out of "`x>100`" as p-uses enables us to track whether both the "`then`" and the "`else`" clauses have been executed.

Second, a subtle difference between these definitions of du-associations and their analogous definitions for imperative programs is that they do not require that a path exist from the definition to the use in the CRG that is

"definition-clear"; that is, they do not specify that the defined cell cannot be redefined ("killed") between the definition and the use. In fact, the definitions do not require a reference to "paths through the CRG" at all. Such requirements are not necessary in the pure spreadsheet languages that we are considering, because in the absence of state-modifying constructs, multiple definitions of a cell do not arise along any particular path in the CRG. This difference is one factor that facilitates more efficient analysis of spreadsheets than of imperative programs, as will be seen in Section 3.1.[5]

Third, even though our CRG is not a control-flow graph, our definition of du-associations, like the definition of du-associations for imperative programs in Rapps and Weyuker [1985], specifies *static du-associations*, defining these associations in terms of a graph representing a program. In imperative programs, such du-associations can be determined by standard static reaching-definitions data flow analysis such as that performed by many compilers; in spreadsheets, a similar approach applies. However, in imperative programs, not all static du-associations are *executable*: there may be no assignment of input values to a program that will cause a definition of variable v to reach a particular use of v. Determining whether such du-associations are executable is provably impossible in general and frequently infeasible in practice [Frankl and Weyuker 1988; Weyuker 1993]; thus, data flow test adequacy criteria typically require that test data exercise (cover) only executable du-associations. In this respect, our criterion (as we shall show) is no exception. Spreadsheets, though typically "simpler" in many ways than programs in imperative languages, can indeed contain nonexecutable du-associations; we further examine issues involving nonexecutable du-associations in spreadsheets in Section 4.4.3. In the rest of this article, to distinguish the subset of the static du-associations in a spreadsheet that are executable, we refer to them as *executable du-associations*.

We next require a definition of what it is for a test case to exercise (cover) a du-association. This definition in turn requires (1) a definition of what it is for a test case (when applied) to "execute" that du-association, and (2) a definition of what it is for a du-association to "influence a cell output."

Toward (1), we first define an *active definition* as the definition, in a cell's formula graph, that was executed (and thus, contributed the current value to the cell that contains it) when that cell was last evaluated. Similarly, we define an *active use* as a use, in a cell's formula graph, that was executed (and thus, contributed to the computation of the current value in the cell) when that cell was last evaluated. (If the use is a p-use (n_2, n_3), this implies that the predicate node n_2 was evaluated such that n_3 was the next node executed.) Let $t = (I, C_2)$ be a test case for spreadsheet S. A du-association (n_1, n_2, C_1) (or $(n_1, (n_2, n_3), C_1)$) in S is *executed* by t if the

[5]Nevertheless, to accommodate spreadsheets containing "impure" features such as imperative-language macros, the definitions *can* be stated in terms of "paths through CRGs" and "definition-clear paths," and thereby support dataflow testing of those spreadsheets.

presence of input values I in the input cells in S caused (dynamically) n_1 and n_2 (or n_1 and (n_2, n_3)), to be active. Note that the absence of definitions that "kill" other definitions, and the dependence-driven evaluation model in spreadsheets, implies that if n_1 and n_2 (or n_1 and (n_2, n_3)) are both active, n_1 "reaches" n_2 (or n_1 and (n_2, n_3))—that is, n_1 has provided the current definition of C_1 that is used as a value for C_1 in n_2 when the expression containing n_2 is evaluated.

Next we consider (2), what it is for a du-association to "influence a cell output." Informally, for a du-association to influence the output of cell C_2, it must directly or transitively contribute to the computation of the definition of C_2. More formally, let S be a spreadsheet containing du-association (n_1, n_2, C_1) (or $(n_1, (n_2, n_3), C_1)$); we say that (n_1, n_2, C_1) (or $(n_1, (n_2, n_3), C_1)$) *directly influences* the output of cell C_2 if n_1 and n_2 (or n_1 and (n_2, n_3)) are active, and if C_2 is the cell containing n_2 (or (n_2, n_3)). We then say (recursively) that (n_1, n_2, C_1) (or $(n_1, (n_2, n_3), C_1)$) *influences* the output of cell C_2 if (n_1, n_2, C_1) (or $(n_1, (n_2, n_3), C_1)$) directly influences the output of C_2, or if (n_1, n_2, C_1) (or $(n_1, (n_2, n_3), C_1)$) influences the output of some cell C_3 that directly influences the output of C_2.

Next, we say that test case $t = (I, C_2)$ *exercises* (covers) du-association (n_1, n_2, C_1) (or $(n_1, (n_2, n_3), C_1)$) if the following three conditions hold: (i) t executes (n_1, n_2, C_1) (or $(n_1, (n_2, n_3), C_1)$), (ii) (n_1, n_2, C_1) (or $(n_1, (n_2, n_3), C_1)$) influences the output of C_2, and (iii) the output of C_2 is explicitly judged by the user to be correct under input configuration I.

Note that the CRG edges are not explicitly used in these definitions, since the most straightforward definitions are directly in terms of the relationships involved. However, other adequacy criteria that relate directly to CRG edges are also possible, and these might be defined most easily in terms of CRG edges.

Given the above definitions, we can define our adequacy criterion. Let S be a spreadsheet, and let T be a test suite for S. T is *OI-All-du-adequate* for spreadsheet S if and only if, for each executable du-association x in S, there exists at least one test case $t = (I, C) \in T$ that exercises x. For brevity, in the rest of this article we abbreviate "*OI-All-du-adequate*" as "*du-adequate*."

3. A METHODOLOGY FOR TESTING SPREADSHEETS

In Section 1, we described four classes of differences between the spreadsheet language paradigm and traditional imperative paradigm. To accommodate these differences, we have developed a testing methodology based on the use of the du-adequacy criterion that is incremental, performing analysis and collecting the data required for testing as the program is being created. We have integrated support for this methodology into the spreadsheet environment at a fine granularity, providing the following functionalities:

—The ability to incrementally determine the (static) du-associations in an evolving spreadsheet whenever a new cell formula is entered.

—The ability to automatically track execution traces, which provide the information necessary to determine the executable du-associations that currently influence output values.

—A user-accessible facility for pronouncing outputs "validated" at any point during spreadsheet development, and the abilities both to determine the du-associations that should be considered exercised as a result of this validation and to immediately communicate to the user how well exercised the visible section of the spreadsheet is.

—The ability to determine the (static) du-associations affected by a spreadsheet change, and immediately depict their altered validation status in the visible section of the spreadsheet.

—The ability to recalculate (static) du-associations and validation information when an entire preexisting spreadsheet is loaded, or when a large portion of a spreadsheet is modified by a single user action.

We next discuss how our methodology provides these functionalities to spreadsheet languages. We present the material in the context and sequence of an integrated spreadsheet development-and-testing session.

3.1 Task 1: Collecting Du-Associations

Suppose, starting with an empty spreadsheet, that the user begins to build the `Clock` application discussed in Section 2.1 by entering cells and formulas, reaching the state shown in Figure 6. Assume that the user does not change any formulas, but simply continues to add new ones. (We remove this restriction later.)

Because it would be expensive to exhaustively compute the du-associations for the entire spreadsheet after each new formula is added, in our methodology these are computed incrementally. Several algorithms for incremental computation of data dependencies exist for imperative programs (e.g., Marlowe and Ryder [1990] and Pollock and Soffa [1989]), and we could adapt one of these algorithms to our purpose. For example, we could place all definitions and uses in the new formula on a worklist, and propagate them forward and backward along CRG edges. However, there are two attributes of spreadsheet systems that allow a more efficient approach.

First, as described in Section 2.4, in spreadsheet languages the syntax of cell formulas ensures that all definitions of C appear in C's own formula, and none of these definitions may be "killed" by any other definition. Second, in spreadsheet systems, the evaluation engine must be called following each formula edit to keep the display up-to-date, visiting at least all cells that directly reference the new cell (which we will term the *direct consumers* of the new cell) and all visible cells that are directly referenced by (are the *direct producers* of) the new cell. At this time, the engine can

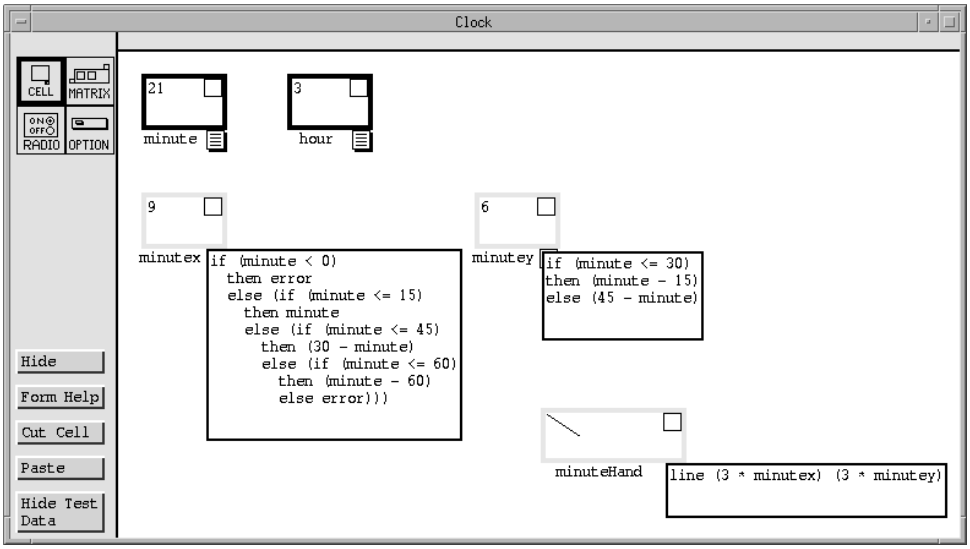

Fig. 6. `Clock` at an early stage. Part of the spreadsheet has been entered.

record *local definition-use information* for the new cell, that is, the definitions and uses that are explicit in the cell's formula. Together, these two attributes mean that (static) du-associations can be incrementally collected following the addition of a cell C simply by linking all definitions in C with all uses of C in direct consumers of C, and linking all definitions in direct producers of C with all uses of those cells in C.[6]

A hash table can efficiently store the following data for each cell C: *C.DirectConsumers*, the cells that reference C; *C.DirectProducers*, the cells that C references; *C.LocalDefs*, the local definitions in C's formula; *C.LocalUses*, the local uses in C's formula; *C.ValidatedID* and *C.UnValidatedID*, integer flags whose uses are described later; *C.DUA*, a set of pairs (*du-association*, *exercised*) for each du-association (d, u) such that u is in *C.LocalUses*, and such that *exercised* is a boolean that indicates whether that association has been exercised; *C.Trace*, which records dynamic trace information for C; and *C.ValTab*, which records validation status. It is reasonable to rely on the formula parser and the evaluation engine to provide the first four of these items, because they are already needed to efficiently update the display and cached value statuses after each edit. The remaining items can be calculated by the testing subsystem.

Algorithm `CollectAssoc` of Figure 7 is triggered when a new formula is added, to collect new du-associations. Lines 2–5 collect du-associations involving uses in C. Lines 6–9 collect du-associations involving definitions (of C) in C. For example, referring back to Figure 6, suppose that the most

[6]See Marlowe and Ryder [1990] for a different view of incremental computation of du-associations as applied within the imperative language paradigm.

```
1.  algorithm CollectAssoc(C)
2.  for each cell D ∈ C.DirectProducers do
3.      for each definition d (of D) ∈ D.LocalDefs do
4.          for each use u of D ∈ C.LocalUses do
5.              C.DUA = C.DUA ∪ {((d,u),false)}
6.  for each cell D ∈ C.DirectConsumers do
7.      for each use u of C ∈ D.LocalUses do
8.          for each def d (of C) ∈ C.LocalDefs do
9.              D.DUA = D.DUA ∪ {((d,u),false)}
```

Fig. 7. Algorithm for Task 1, collecting (static) du-associations.

recent formula entered is that for cell `minutey`. Note that its value is displayed, even though the spreadsheet has not been completely entered; when the evaluation engine is triggered to display this value, it collects *C.DirectConsumers*, *C.DirectProducers*, *C.LocalDefs*, and *C.LocalUses* for `minutey` (as done previously for the other cells on display when their formulas were entered). Called with cell `minutey`, `CollectAssoc` employs this information to collect six new du-associations, described using the node numbers of Figure 5 as: (2,(19,20),`minute`), (2,(19,21),`minute`), (2,20,`minute`), (2,21,`minute`), (20,50,`minutey`), and (21,50,`minutey`).

`CollectAssoc` runs in time $O(ud(DirectProducers + DirectConsumers))$, where *DirectProducers* and *DirectConsumers* are the number of direct producers and direct consumers respectively of C, and u and d are the maximum number of uses and definitions per cell, respectively, in those cells. In practice, u and d are small, bounded by the maximum length of a single formula, which is constant-bounded in most spreadsheet languages. In this case the algorithm's time complexity is $O(DirectProducers + DirectConsumers)$.

3.2 Task 2: Tracking Execution Traces

To track execution traces, which enable the incremental computation of du-associations that have been exercised, it is sufficient to insert a probe into the evaluation engine. When cell C executes, this probe records the execution trace on C's formula graph, storing it in *C.Trace*, adding only O(1) to the cost of execution. For example, in the case of `Clock`, at the moment depicted in Figure 6, the execution trace stored for cell `minutey`, described in terms of Figure 5's node numbers, is (18,19,20,22). If the cell is subsequently reevaluated, the old execution trace is replaced with the new one. Storing only the most recent execution trace in *C.Trace* is sufficient for coverage computation because the cumulative coverage in *C.DUA* is updated incrementally during validation, as we shall describe in our discussion of Task 3.

3.3 The Impact of Tasks 1 and 2 on Spreadsheet Efficiency

The importance of the time complexity of our testing methodology should not be underestimated, because unlike previous approaches to automated

testing support, the costs of our testing algorithms are borne by the user during *program entry* and *execution*. Thus, for viability with spreadsheet users, it is important for any costs that are noticeable to always result in immediate, visible benefits.

Tasks 1 and 2 are necessary preparatory tasks for user testing, but they do not themselves produce any visible benefits for users. Thus, their impact on efficiency is critical in two ways: First, if these "rewardless" preparatory tasks slowed the user down in entering formulas or seeing results, the user would probably simply turn off the testing subsystem. Second, if implementing this testing methodology in a spreadsheet language precluded use of a particular evaluation strategy or of value-caching optimizations, it might not be viable for commercial use.

Task 1 is invoked whenever a new formula is typed in. Its time complexity of $O(DirectProducers + DirectConsumers)$ is no more than the order required by most evaluation engines' cell traversal needed to maintain a correct display and process cached values when a new formula is added—the event that triggers `CollectAssoc`. To see why this is true, consider the possible evaluation engine strategies. As discussed in Section 2.2, the two overall strategies possible are eager evaluation, in which the new formula's result is "pushed" to the cell's consumers, and lazy evaluation, in which visible cells "pull" results from their producers when needed for output. Hence, eager evaluation must read the direct producers in order to compute the result of the new formula, and must access the consumers (both direct and transitive) in order to push the new result along (which also ensures that the value cache is up-to-date).

Lazy evaluation also must read the direct producers in order to compute the new result and display it, because a cell must be on the screen to be edited. However, under lazy evaluation there are several possibilities for how to manage the cache information, and this determines whether the direct consumers will be accessed. The most widely used value-caching mechanism with lazy evaluation is *lazy evaluation with eager marking* (LazyEM), in which consumers of a newly computed result are eagerly marked out-of-date (but not actually recomputed until needed, because evaluation itself is lazy).[7] LazyEM thus, like eager evaluation, accesses all consumers (both direct and transitive).

Although these are the two evaluation approaches widely used for languages in this class, and both require at least $O(DirectProducers + DirectConsumers)$ time, other approaches are possible that instead access consumers visible on the screen (both direct and transitive), some of whose bounds are greater and some fewer than the total number of visible and nonvisible direct consumers. See Burnett et al. [1998] for a description and performance comparison of eager evaluation and LazyEM, as well as several other variations on lazy evaluation, in the context of a class of

[7]The LazyEM algorithm is formally presented and analyzed in Hudson [1991], in which it is shown to be optimal in the number of computations performed, although not optimal in overhead.

visual languages that includes spreadsheet languages. The presence of these possibilities shows that O(*DirectProducers* + *DirectConsumers*) is not a lower bound for spreadsheet evaluation and cache maintenance. However, we have found reports only of eager evaluation and LazyEM being actually used to respond to the trigger of a new formula entry, and O(*DirectProducers* + *DirectConsumers*) is less than their time costs. This allows the cost of collecting the du-associations to potentially be masked by the cost of evaluation activity, at least under these two widely used evaluation approaches.

Regarding Task 2, we have already pointed out that this task is accomplished simply by inserting a probe into the evaluation engine, at a cost of O(1), so we turn our attention to the issue of compatibility (in terms of correctness) with various evaluation strategies. The following discussion explains why our approach to Task 2 functions correctly for all varieties of evaluation engines—i.e., whether the engine eagerly or lazily evaluates cells, following any dependence-preserving evaluation sequence, all cells have associated with them their most recent execution trace.

The correctness of this approach for any evaluation engine rests upon two attributes of this strategy. First, each cell's execution trace is a set rather than a list, rendering any difference in order of execution irrelevant. Second, the granularity of the nodes in the formula graph (and hence in the execution trace) is coarser than the level at which subtle differences among evaluation approaches occur.

To illustrate the second attribute, consider execution of C. C's entry point and exit point (both of which are unique) are by definition always executed. If C's interior (nonentry/exit) nodes do not contain any conditionals ("if" expressions) then there is only one interior node, which contains the entire expression and which, because it is executed *at least in part*, is recorded in the trace. On the other hand, if there are conditionals, the strategy is the same eager or lazy—execute the condition node and either the "then" or the "else" node but not both. (Although it may seem that eager evaluation would execute both the "then" and the "else," this could lead to runtime errors, and hence even eager approaches employ "short circuit" evaluation to execute conditionals. An example of a formula that must be executed lazily to avoid such runtime errors is "if A1=0 then 0 else 10/A1.")

Note, that, due to the fact that the execution traces are stored for each cell, an evaluation engine's particular caching optimizations do not change the execution traces. If cell D is executed because cell C's execution requires it, there are only two possibilities: either D has a cached value or it does not. If D has a cached value, then D's stored execution trace is still up-to-date. If D does not have a cached value, then D will be executed and its execution trace stored. In neither case does the execution of D affect the execution trace of C. Hence, whether D was executed now or at some previous time does not change the trace stored for C or the trace stored for D.

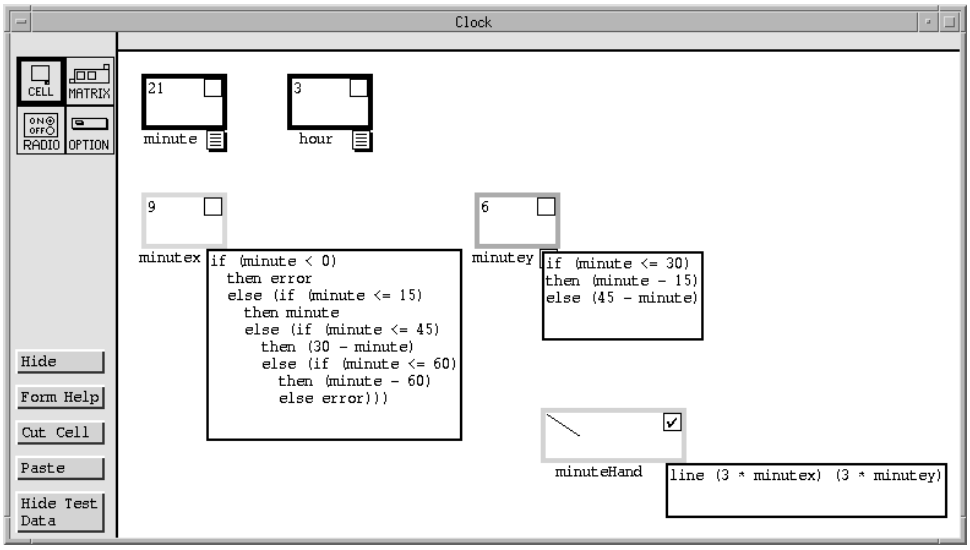

Fig. 8. The evolving `Clock` spreadsheet at an early stage, after the `minuteHand` cell has been validated.

3.4 Task 3: Pronouncing Outputs "Validated"

In this section, we show how the data collected in Tasks 1 and 2 can be used to provide test adequacy information to the user in a way that requires no understanding of formal notions of testing, and uses visual devices to draw attention to untested sections of the evolving spreadsheet.

In the desktop clock programming scenario, suppose that the user looks at the values displayed on the screen and decides that the `minuteHand` cell contains the correct value. To document this fact, the user clicks on the validation tab in the upper right corner of that cell. As Figure 8 shows, one immediately visible result of this action is the appearance of a checkmark in the validation tab. If the user enters another input in cell `minute`, `minuteHand`'s validation checkmark changes to a question mark as in Figure 9, which means the current value has not been validated but some previously displayed value has. (Any evaluation engine must visit at least on-screen consumers of the new input to keep the displayed values up-to-date, so changing the checkmark to a question mark during these visits adds only O(1) to the cost of each.) The third possible appearance, a blank validation tab, means no validations have been done since the last formula change to C or to a noninput cell affecting C. Thus, the validation tab keeps the user apprised of which cells have been explicitly validated and which have not, given the current collection of formulas.[8]

[8]Of course, in validating a cell output, a user may make an incorrect judgment. The "oracle problem" [Weyuker 1982] of determining whether or not an output is correct is significant, and with spreadsheets, where specifications typically do not exist, may be less amenable to solution than with traditional imperative programs. This problem, however, exists widely, and is also present with the ad hoc approaches currently utilized by spreadsheet users.

Propagating the implications of the user's validation involves test adequacy. Whenever a du-association influences the production of a validated value, the *exercised* flag for that du-association (the second item of data kept for each du-association in the *.DUA* set for the cell in whose formula the use occurs) is set to "`true`." The percentage is then calculated of the du-associations, whose uses occur in the cell, that have been exercised. This percentage is used to determine the cell's border color on a continuum from red (untested) to blue (100% of the du-associations whose uses occur in the cell having been exercised). (In this black-and-white article, the continuum is light gray to black.) With each validation that exercises a previously unexercised du-association, the border becomes less red (darker in these figures), indicating a greater degree of "testedness" for that cell. This visual feedback appears in all cells that contributed to the computation of the value in the validated cell.

In the example shown in Figure 8, the computation of `minuteHand`'s validated value involved two of the four du-associations that end in `minutey`, two of the seven du-associations that end in `minuteHand`, and four of the 13 du-associations that end in `minutex`. Thus, because the user validated `minuteHand`, the cell borders were darkened using these fractions. Input cells are, by definition, fully exercised. When borders are entirely blue, the user can see that each cell reference pattern (du-association) has been tested (i.e., exercised with validation) at least once.

At the point reached in Figure 8, the user has been performing tasks (entering formulas and validating) at the granularity of cells, which is the usual granularity for spreadsheet tasks. But, as we have pointed out, the system's reasoning about testedness is actually taking place at the granularity of du-associations (patterns of cell references, in spreadsheet users' terminology). Figure 9 shows how the system can provide help to a user puzzled as to how to achieve the desired blue borders, by explicitly depicting which cell reference patterns (du-associations) still need to be tested.

As the figure shows, the user can display arrows for any cell, in order to see cell reference patterns (du-associations). These arrows point from definitions to uses and are colored using the same scheme as the borders. For example, the four du-associations involving uses of minute in `minutey` are depicted by three arrows. The top in-arrow represents definition-p-use associations (2,(19,20),`minute`) and (2,(19,21),`minute`), one of which has been exercised and one of which has not; hence that arrow is purple (50% blue and 50% red). The middle in-arrow represents definition-c-use association (2,20,`minute`), which was exercised by the previous validation. The bottom in-arrow, representing definition-c-use association (2,21,`minute`), is red because it has not yet been exercised. The user can see arrows at this granularity for any visible formula. When the user undisplays a formula, as for cell `minutex`, du-association arrows pointing into and out of that cell coalesce, thereby summarizing at the granularity of cells. The fact that the arrows are colored makes explicit which cell reference patterns still need to be tested. As the red (pale gray) bottom arrow into cell `minutey` shows, one

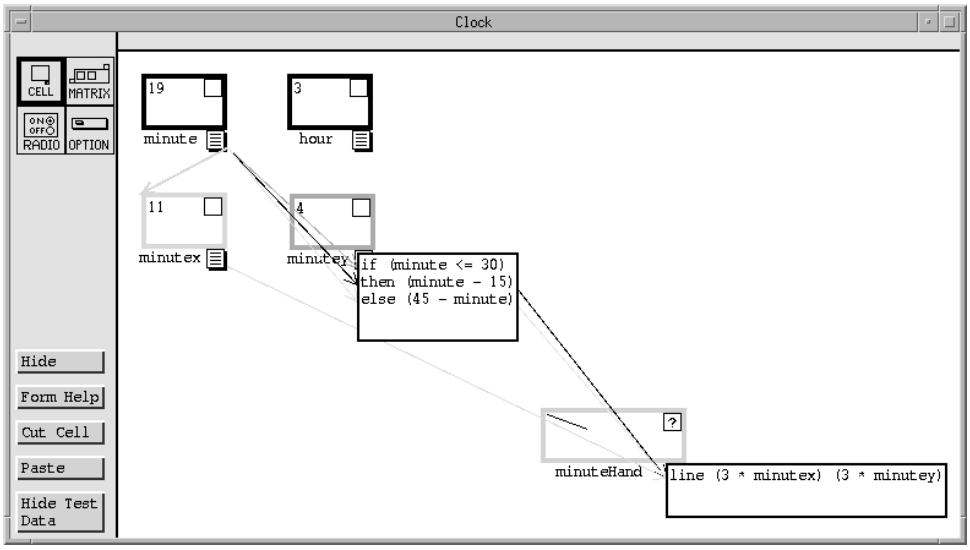

Fig. 9. The evolving `Clock` spreadsheet after a new value has been entered into the Minute cell, and after the user has clicked on cells to display their arrows.

way to darken `minutey`'s border color would be for the user to next choose a value of `minute` that exercises the "else" clause in `minutey`, and then validate one of the cells influenced by the du-association from `minute` to the "else" clause.

Figure 10 displays our algorithm `Validate`, which is invoked when the user pronounces a displayed value valid. The algorithm uses the du-association information and execution traces, previously calculated and stored as discussed in the descriptions of Tasks 1 and 2, to calculate the du-associations that influence *C*'s current value, and to update borders of participating cells.[9] As the algorithm proceeds, it adds to stored *.DUA* data that indicates the du-associations that have influenced validated cells thus far. This coverage information is accumulated and retained across a succession of tests, even though cell execution traces change as subsequent tests are applied.

Variable *ValidatedID*, referenced in the algorithm, can be set to 0 when the spreadsheet environment is first activated. Then, when cells are created or added to the spreadsheet, their *.ValidatedID* fields are initialized to 0. On each invocation of `Validate`, *ValidatedID* is incremented

[9]A generalization of this algorithm related to the approach of Duesterwald et al. [1992] uses slicing to locate the expressions that contribute to the computation of the validated output, and identifies the du-associations involved in the computation from that slice. This generalized approach is applicable to languages with recursion, iteration, and redefinitions of variables.

```
1.  algorithm Validate(C)
2.  ValidatedID = ValidatedID +1
3.  C.ValTab = "checkmark"
4.  ValidateCoverage(C)
5.  procedure ValidateCoverage(C)
6.  C.ValidatedID = ValidatedID
7.  for each use u ∈ C.Trace do
8.      D = the cell referenced in u
9.      d = the current definition of D found in D.Trace
10.     C.DUA = C.DUA ∪ {((d,u),true)} - {((d,u),false)}
11.     if D.ValidatedID < ValidatedID then
12.         ValidateCoverage(D)
13. UpdateDisplay(C)
```

Fig. 10. Algorithm for Task 3, updating test adequacy information following a validation action.

(line 1). The .*ValidatedID* fields for all cells visited are assigned this value of *ValidatedID*, which prevents duplicate visits to the same cell.[10]

The use of *ValidatedID* ensures that `ValidateCoverage` is called no more than once per cell, and that `Validate` terminates in worst-case time proportional to the number of du-associations that have influenced a validated cell. Because the set of uses in a cell's trace corresponds to a set of definitions in that cell's direct producers, which in turn lead to that cell's indirect producers, the cost of validation is bounded by the number of direct and transitive producers of a cell. This is less than or equal to the cost of calculating the cell's value the first time (when no reusable values are present in the cache). However, the algorithm is triggered by a user interaction that does not require evaluation, so, unlike the other algorithms we have presented, its cost cannot be masked by the cost of the evaluation process.

3.5 Task 4: Adjusting Test Adequacy Information

So far, we have focused on how our methodology handles cell formulas as they are added to a spreadsheet. We now consider the other basic edits possible with spreadsheets, namely, deleting a cell or changing a cell's formula. Changes to a constant-formula cell are equivalent to the application of a new test input (which may or may not be followed by validations by the user), and require no action beyond that involved in recalculating execution traces as discussed under Task 2. Deletion of a cell is equivalent to modifying that cell's formula to BLANK. Thus, we need only consider modifications to nonconstant formulas.

Suppose that the user has done quite a bit of testing, and has discovered a fault that requires a formula modification with far-reaching consequences. The user may believe that the spreadsheet is still fairly well

[10]By using an integer rather than a boolean, and incrementing it on each invocation of the algorithm, we avoid the need to initialize the flag for all cells in the spreadsheet on each invocation. We assume that *ValidatedID* will not overflow, to simplify the presentation.

```
1.  algorithm UnValidate(C)
2.  global AffCells = {}
3.  UnValidatedID = UnValidatedID +1
4.  UnValidateCell(C)
5.  for each cell D ∈ AffCells do
6.      UpdateValTab(D)
7.      UpdateDisplay(D)
8.  procedure UnValidateCell(E)
9.  E.UnValidatedID = UnValidatedID
10.  AffCells = AffCells ∪ E
11.  for each cell F ∈ E.DirectConsumers do
12.    for each definition d (of E) in E do
13.      for each ((d,u),true) ∈ F.DUA do
14.        F.DUA = F.DUA ∪{((d,u),false)}−{((d,u),true)}
15.    if F.UnValidatedID < UnValidatedID then
16.      UnValidateCell(F)
```

Fig. 11. Algorithm for Task 4, updating test adequacy information following a modification.

tested, and not realize the extent to which the modification invalidates previous testing.

To address this lack of awareness, a system must immediately reflect the new test adequacy status of the spreadsheet whenever a cell is modified.[11] To accomplish this, the system must (1) update *C*'s du-association and execution trace information, and (2) update the *exercised* flags on all du-associations that may be affected by the modification, allowing calculation and display of new border and arrow colors to reflect the new "testedness" status of the spreadsheet. Validation tab statuses on visited cells must also be adjusted, changing all checkmarks to question marks if the cell retains any exercised du-associations after affected du-associations have been reset, or to blank if all the cell's exercised flags are now unset. For example, in the completed `Clock` spreadsheet, if the user changes cell `minutex`'s formula, then the du-associations involving `minutex`, and the validation statuses for `minutex`, `minuteHand`, and `theClock`, must all be adjusted.

Our methodology handles item (2) first, removing the old information before adding the new. Let *C* be the modified cell. We use a conservative approach that recursively visits potentially affected cells. The algorithm, `UnValidate`, given in Figure 11, is similar to `Validate`, but instead of using dynamic information to walk backward through producers, it uses static information to walk forward through consumers. As the algorithm walks forward, it changes the *exercised* flag on each previously exercised

[11]In this context, the problem of interactive, incremental testing of spreadsheets resembles the problem of regression testing imperative programs, and we could adapt techniques for incremental dataflow analysis (e.g., Marlowe and Ryder [1990] and Pollock and Soffa [1989]) and incremental dataflow testing of imperative programs (e.g., Gupta et al. [1996], Harrold and Soffa [1988], and Rothermel and Harrold [1994]) to generalize our approach. This generalized approach would apply to spreadsheet languages in which cell references can be recursive or in which formulas contain iteration. In the absence of such features, the simpler approach that we present here suffices.

du-association it encounters to "false," and keeps track of each cell visited in *AffCells*. On finishing the work for all the cells, the algorithm updates the border and arrow colors and validation tab for each cell in *AffCells*.

At this point, the du-association and trace information stored with C can be updated. First, all stored du-associations involving C are deleted; these du-associations are found in the information stored for C and for cells in *C.DirectConsumers*. This deletion also guarantees that du-associations that end in C are no longer marked "exercised." Having removed the old du-associations, it is necessary only to reinvoke CollectAssoc as described in Section 3.1 to add new associations. Finally, stored execution traces are automatically updated via the evaluation engine as described earlier.

Because UnValidate's processing is consumer-driven, then as with Task 1, the cell visits required by it are already required for display and value cache maintenance under eager evaluation and under LazyEM, but not necessarily by other evaluation engines that may be possible. However, in the cases of eager evaluation and LazyEM, the time cost of the algorithm increases only by a constant factor the cost of other work being performed by the environment when a formula is edited.

3.6 Task 5: Batch Computation of Information

Test information can be saved when a spreadsheet is saved; then, when the spreadsheet is reloaded later for further development, it is not necessary to analyze it exhaustively to continue the testing process. Still, there are some circumstances in which it may be necessary to calculate du-association information for a whole spreadsheet or section of a spreadsheet, such as when the user does a block copy/paste of cells, or imports a spreadsheet from another environment that does not accumulate the necessary data. One possible response to such an action is to iteratively call the CollectAssoc algorithm presented earlier for each cell in the new spreadsheet section. This approach, however, may visit cells more times than necessary.

Figure 12 presents a more efficient approach, BatchCollectAssoc, that takes an entire set U of cells as input, collects (from the spreadsheet environment) the set V of cells that are direct producers of cells in U, and then makes a single pass over V to update information on du-associations and validation status.[12] Although this algorithm has the same worst-case runtime as CollectAssoc, when there are interrelationships among the cells its set-driven approach allows it to eliminate some duplicated visits to cells.

[12]Another approach to this problem is to propagate definitions forward, and uses backward, across flow and cell dependence edges in the CRG; this approach applies to spreadsheet languages in which cell references can be recursive or in which formulas contain iteration or redefinitions of variables. In the absence of such features, the simpler approach that we present here is more efficient.

```
1.  algorithm BatchCollectAssoc(U)
2.  V = {}
3.  for each cell C ∈ U do
4.      V = V ∪ C.DirectProducers
5.  for each cell C ∈ V ∪ U do
6.      for each cell D ∈ C.DirectConsumers do
7.          for each definition d (of C) ∈ C.LocalDefs do
8.              for each use u of C ∈ D.LocalUses do
9.                  D.DUA = D.DUA ∪ {((d,u),false)}
```

Fig. 12. Algorithm for Task 5, batch collection of (static) du-associations.

3.7 Visual Representation Devices

The visual representation of testedness used in this article reflects three constraints on the visual representation that we believe to be important for integration into spreadsheet environments. We derived these constraints using representation design metrics drawn from literature on cognitive aspects of programming [Green and Petre 1996; Yang et al. 1997]. The constraints we placed upon the visual representation are that first the visual representation should be frugal enough of screen space that it does not significantly decrease the number of cells that can be displayed. Second, the visual representation should maintain consistency with the current formulas in the visible cells. Third, the visual representation devices should be accessible to as wide an audience as possible, including those with mild visual impairments.

In the prototype used to create the figures in this article, we used the following representation devices to satisfy these constraints. To satisfy the first constraint, we chose to encode testedness information with cell border colors rather than display it in textual summaries, and to encode validation status with the presence or absence of checkmarks or question marks. The auxiliary colored arrows are optional and transient; they can be displayed or undisplayed by clicking on a cell, and do not permanently occupy screen space.

To satisfy the second constraint, the display of testedness is automatically updated whenever any action by the user or by the system changes testedness status. Thus, outdated testedness information is never left on the screen.

To satisfy the third constraint, we selected border colors along a red-blue continuum to be used against the usual white background for spreadsheets. The colors red and blue are easily discriminated [Christ 1975] and because, due to the physiology of the human eye, red stands out while blue recedes [Shneiderman 1998], they further our goal of drawing users' attention to untested cells. The blue can also be desaturated to enhance this effect. Also, because red and blue differ in two of the three RGB components of screen color, this device should be usable by some red-deficient or blue-deficient users [Murch 1984]; the gray-black continuum of this article is also an option for color-deficient users.

An additional visual device we employed to emphasize the differences among the three categories of no testedness, partial testedness, and complete

testedness was to use a quadratic formula that separates the 0% and 100% colors from the partial testedness colors. Of course, while this furthers the goal of drawing overall attention to untested cells, the exact degree of testedness for a particular cell is not likely to be readily discernible by most users. We are considering an additional optional thermometer-like indicator along a border for when a user wishes to see more exact information about a cell's testedness.

An issue we have yet to address is whether users will confuse "testedness" (as depicted with colors) with "correctness." We suspect that this is likely, because it occurs in other kinds of informal testing situations. A possible way to communicate the difference between these two concepts might be to let the user select different levels of testing to try, ranging from "novice" to (unachievable) "guaranteed correctness." These could be implemented behind the scenes via a progression of increasingly stronger adequacy criteria. For example, a novice level could be implemented via a very simple criterion such as cell adequacy, some higher level implemented via du-adequacy, and so on. The "guaranteed" highest level could then be implemented via a message saying that testing can never actually guarantee that all of the faults have been revealed. User studies are planned to further investigate the effectiveness of these devices for communicating effectively about testedness.

4. EMPIRICAL STUDY

Our visual feedback is designed to help users achieve du-adequate testing, which is what is needed for borders to turn blue. However, we have not yet presented evidence that du-adequate testing will reveal a reasonable percentage of faults in spreadsheets, or that it will function any more effectively than a random testing approach. These are fundamental issues to study, because if du-adequate testing does not detect faults effectively there may be no reason to pursue it further. To address these issues, we consider the following research questions.

Q1: How effective are du-adequate test suites at revealing faults in spreadsheets?

Q2: How do du-adequate test suites compare to randomly created test suites of the same size in terms of their effectiveness at revealing faults in spreadsheets?

4.1 Measures

A test suite's efficacy is a measure of its ability to detect faults. In general, a test suite's efficacy cannot be measured directly, and is often approximated by various coverage or adequacy measures. In our experiments, however, we work with spreadsheets that contain single known faults; thus, test suite efficacy can be measured by determining how many of these known faults a test suite can detect.

Table I. Experiment Subjects

Spreadsheet Name	No. of Cells	No. of Expressions	No. of Du-Associations	No. of Faulty Conditions	No. of Versions	Test Pool Size	Avg. Test Suite Size
Clock	13	33	64	7	7	250	11.3
Digits	7	35	89	14	10	230	22.7
FitMachine	9	33	121	12	11	367	30.2
Grades	61	61	55	0	10	80	9.8
MicroGen	6	16	31	5	10	170	10.4
Sales	30	30	28	0	9	176	10.4
Solution	7	20	32	7	11	99	12.0
TimeCard	12	33	92	12	8	240	16.7

More precisely, let S be a spreadsheet, and let $S_F = \{S_1, S_2, \ldots, S_k\}$ be a set of k faulty versions of S, each containing a single fault. Let T be a test suite for S, that detects j $(j \leq k)$ of the faults in the versions in S_F. The *efficacy of* T is the percentage of faulty versions in S_F whose faults are detected by T, and is given by $j/k*100$.

This efficacy measure requires a definition of what it is for a fault to be detected by a test suite. Recall that we define a test case t to be a tuple (I, C), where I is an assignment of values to input cells, and C is a cell whose output value has been examined by the user for correctness. Given a correct spreadsheet S, faulty version S_j of S, and test suite T for S, we say that T detects the fault in S_j if there exists some test case $t = (I, C) \in T$ that, applied to S and S_j, causes cell C to display a different value in S_j than in S.

This measure lets us measure and compare test suites' ability to reveal faults in spreadsheets. Similar measures have been used in previous studies of the efficacy of test suites created by various test adequacy criteria [Frankl and Weiss 1993; Hutchins et al. 1994].

4.2 Experiment Instrumentation

4.2.1 *Subject Spreadsheets.* Because our methodology has been implemented within the Forms/3 programming environment, that was the environment used for our study. In keeping with this fact, for our study we obtained eight Forms/3 spreadsheets from experienced Forms/3 users. Table I lists details about these spreadsheets. Three of the spreadsheets—TimeCard, Grades, and Sales—are modeled after spreadsheets written in commercial spreadsheet systems. FitMachine and MicroGen are simple simulations. Clock is a graphical desktop clock. Digits is a number-to-digits splitter, and Solution is a quadratic equation solver. Table I also lists the numbers of cells, expressions (equivalent to the number of nonentry, nonexit nodes in CRGs), conditions (equivalent to the number of predicate nodes in CRGs) and du-associations (pairs consisting of a definition node and a c-use node, or a definition node and a labeled edge out of a p-use

node, as defined in Section 2.4) in the spreadsheets. Of the spreadsheets, all but Grades and Sales utilized conditional computations.

4.2.2 *Faulty Versions, Test Cases, Test Pools, and Test Suites*. To address our research questions, we required faulty versions of our spreadsheets, du-adequate test suites, and randomly created test suites of the same size. To obtain these, we followed a procedure similar to that utilized by Hutchins et al. [1994] in their study of the fault-detecting abilities of several varieties of test suites applied to imperative programs. The procedure was as follows.

We asked seven users experienced with Forms/3 and commercial spreadsheets, working without knowledge of one another's work, to manually seed faults into our subject spreadsheets which, in their experience, are representative of faults found in spreadsheets. This process yielded between 7 and 11 faulty versions of each subject program, as noted in Table I.

We next asked a Forms/3 user who had no knowledge of these specific faults to generate, for each of the eight nonfaulty subject spreadsheets, a large "test pool" containing possible test cases for these spreadsheets. To populate each test pool, the user first created an initial pool of test cases exercising the spreadsheet's functionality. He then measured the du-adequacy of this initial pool, and augmented the pool to ensure that each executable du-association in the spreadsheet was exercised by at least five test cases in the pool. The user ensured that the test pools contained no duplicate test cases. For each test case, he also verified that validated cells in the original program produced correct values. The resulting test pools ranged in size from 80 to 367 test cases, as shown in Table I.

We next used our test pools to create du-adequate test suites for our spreadsheets. To do this, we first determined, for each test case t in the test pool, the du-associations exercised by t. We then created a test suite T by randomly selecting a test case from the test pool, and adding it to T only if it added to the cumulative coverage achieved by test cases added to T thus far, repeating this process until T was du-adequate. We discarded duplicate test suites. This process yielded between 10 and 15 du-adequate test suites for each of our subject spreadsheets; Table I lists the average sizes of these test suites.

Finally, to create randomly selected test suites, for each spreadsheet S, for each du-adequate test suite T for S, we randomly selected $|T|$ test cases from the test pool for S. This process yielded, for each spreadsheet S, a set of randomly selected test suites of the same sizes as the du-adequate test suites for S.

4.3 Experiment Procedure

The experiment was run using an 8×2 factorial design with between 10 and 15 test suite efficacy measures per cell; the two categorical factors were:

Table II. Mean Test Suite Efficacies for Du-Adequate and Randomly Generated Test Suites, for the Eight Subject Spreadsheets, and p-Values from Wilcoxon Rank-Sum Test

Spreadsheet Name	Mean Test Suite Efficacy (Du-Adequate Suites)	Mean Test Suite Efficacy (Randomly Generated Suites)	p-Values from Wilcoxon Rank-Sum Test
Clock	96.2	74.3	0.0001
Digits	80.0	68.0	0.0053
FitMachine	98.8	81.2	0.0001
Grades	77.0	59.0	0.1724
MicroGen	73.0	44.0	0.0133
Sales	95.5	73.3	0.0028
Solution	70.0	60.0	0.1375
TimeCard	56.7	48.3	0.1454

—The subject spreadsheet (8 programs, each with a variety of faulty versions).

—The variety of test suite utilized (du-adequate or randomly generated).

For each spreadsheet S and each variety of test suite V we applied the following procedure:

for each test suite T of variety V for spreadsheet S
(1) we ran all test cases in T on S, saving outputs,
(2) for each faulty version S_i of S:
  (a) we ran all test cases in T on S_i, saving outputs,
  (b) we recorded that T detected the fault in S_i if the output of the validated cell for some test case t in T executed on S_i differed from the output of that cell when t was executed on S.

We counted the number of times that T had been determined to detect (in step 2(b)) that a version of S was faulty. We then divided this number by the number of faulty versions k and multiplied that result by 100. This yielded our measure of the *efficacy of* T—the percentage of faulty versions of S whose faults were detected by T—as defined in Section 4.1.

4.4 Data and Analysis

4.4.1 *Test Suite Efficacy*. Table II presents the mean test suite efficacies calculated for the eight subject spreadsheets for du-adequate and randomly generated test suites, and the p-values obtained from applying the Wilcoxon Rank-Sum Test to the test suite efficacy data. As the table indicates, on all eight spreadsheets, du-adequate test suites outperformed randomly generated test suites of the same size: differences in average efficacies ranged from 8.4% on TimeCard to 29% on MicroGen. The Wilcoxon test indicates that these differences were statistically significant ($p < 0.05$) on five of the eight spreadsheets.

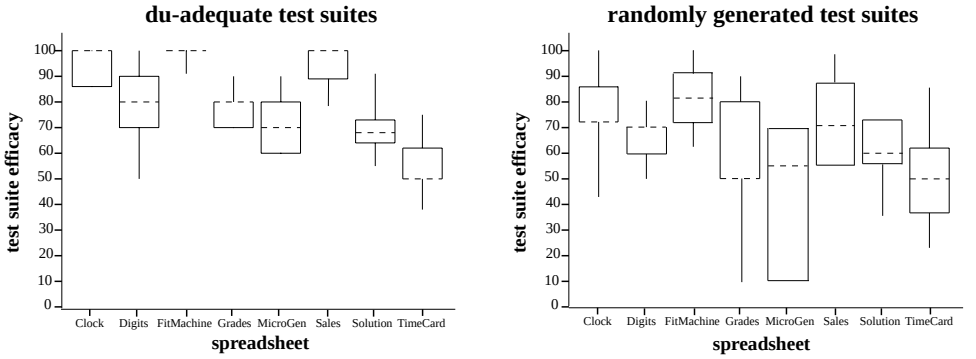

Fig. 13. Test suite efficacy data for du-adequate (left) and randomly generated (right) test suites.

Figure 13 contains boxplots depicting the ranges and medians of these data. The graph on the left presents results for du-adequate test suites; the graph on the right presents results for randomly generated test suites. Each graph contains eight boxplots: one for each of the eight subject spreadsheets studied.[13] The boxplots illustrate wider variances in the efficacies of randomly generated test suites in comparison to the efficacies of du-adequate test suites, on all spreadsheets other than Digits. The boxplots also illustrate that the median efficacies exhibited by randomly generated test suites were lower than those exhibited by du-adequate test suites.

In summary, the efficacy data gathered in the study support the claim that, for the spreadsheets, versions, and test cases utilized, du-adequate test suites generated for the spreadsheets possess higher efficacies than randomly generated test suites of the same size. Further, these du-adequate test suites seem to be more consistent in their ability to reveal faults than their randomly generated counterparts.

4.4.2 *Results per Fault.* Both du-adequate and randomly generated test suites were observed to miss faults in our experiment; thus, we also examined the extent to which the faults in our spreadsheets could be detected by those test suites. Figure 14 depicts relevant data; the figure contains a separate graph for each of the eight subject spreadsheets. In each graph, each faulty version of the spreadsheet occupies a position along the x-axis and is represented by a pair of vertical bars. The two bars depict

[13]A boxplot is a standard statistical device for representing data distributions [Johnson 1992]. In these plots, each data set's distribution is represented by a box. The box's height spans the central 50% of the data, and its upper and lower ends mark the upper and lower quartiles. The middle of the three horizontal lines, depicted as a dashed line, represents the median. The vertical lines attached to the box indicate the tails of the distribution. Consider, for example, the boxplot for `Digits` in the graph of du-adequate test suite results. This boxplot shows that the median test suite efficacy measure among the 10 test suites for `Digits` was 80%, with half of the test suites exhibiting efficacies evenly distributed between 70% and 90%, and with the efficacies of the 10 suites ranging overall from 50% to 100%.

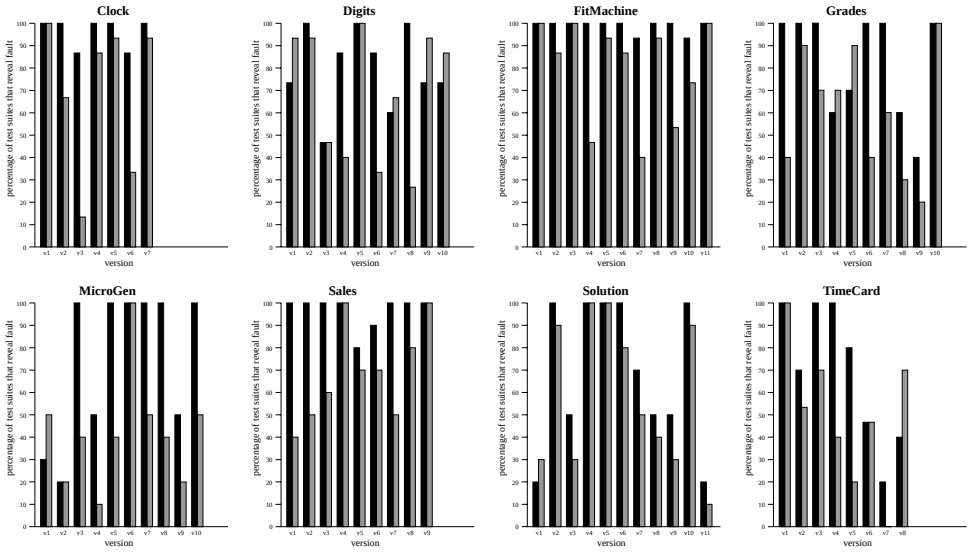

Fig. 14. Percentages of test suites that revealed faulty versions, per spreadsheet, per version. Black bars depict results for du-adequate test suites; gray bars depict results for randomly generated test suites.

the percentage of the du-adequate test suites (black bars) and the percentage of the randomly generated test suites (gray bars) for that spreadsheet, respectively, that detected the fault in that faulty version.

Altogether, our study involved 76 faulty versions. As Figure 14 illustrates, among these versions, there were 51 in which the du-adequate test suites were more successful than their randomly generated counterparts at revealing the faults; there were only 9 in which the randomly generated test suites were more successful than their du-adequate counterparts. On 22 versions, randomly generated test suites were over 50% less successful at revealing the fault than du-adequate test suites; there were no cases in which du-adequate test suites were over 50% less successful than randomly generated suites. There were 44 versions in which the fault was revealed by every du-adequate test suite; there were only 9 of these versions in which the fault was revealed by every randomly generated test suite. There were no versions in which the fault was never revealed by any du-adequate suite. These findings further indicate the greater effectiveness of du-adequate test suites relative to randomly generated test suites of the same size.

It would be useful to be able to characterize the faults that, in our study, were more or less easily detected by du-adequate test suites, as well as those faults that were more easily detected by randomly generated suites than by du-adequate test suites. Inspection of the faults in our programs, however, yielded no obvious, consistent categorization scheme.

4.4.3 *Nonexecutable Du-Associations*. A significant problem in dataflow testing involves (static) du-associations that are recognized by the testing

Table III. Nonexecutable Du-Associations in the Subject Spreadsheets

Spreadsheet Name	Number of Du-Associations	Number of Nonexecutable Du-Associations	Percentage Nonexecutable
Clock	64	9	14.1
Digits	89	28	31.5
FitMachine	121	20	16.5
Grades	55	0	0.0
MicroGen	31	3	9.7
Sales	28	0	0.0
Solution	32	2	6.3
TimeCard	92	8	8.7

system's analysis, but that are nonexecutable. We discovered that 70 (13.7%) of the 512 du-associations in our subject spreadsheets were nonexecutable; Table III provides data on these. The number of nonexecutable du-associations varied widely, however, across spreadsheets: Digits and FitMachine each contained over 20 nonexecutable du-associations; Grades and Sales, which contain no conditional expressions, contained none. The 13.7% overall average rate of nonexecutability among du-associations is lower than the average rates of 26% and 27% observed in two studies of imperative programs reported in Weyuker [1993]; nevertheless, the presence of these du-associations could be difficult to explain to users. Future work will address the possibility of applying techniques such as those of Clarke [1976] to help automatically identify (to the extent possible) these nonexecutable du-associations.

4.5 Threats to Validity

Finally, we discuss several threats to the validity of our experiment. In this experiment, our measurements of test suite efficacy were highly accurate, but test suite efficacy is not the only possible measure of the quality of test suites. For one thing, we have focused on measuring the effectiveness of du-adequate test suites, without measuring their cost. Ultimately, because efficacy only partially captures the aspects of test suite quality in which we are interested, we will need to also consider other measures.

Other threats to validity are centered around the issue of how representative the subjects of our experiments are. More generally, it is reasonable to ask whether our results are dependent on the way in which our spreadsheets, faulty versions, test pools, and test suites were created. Our subject spreadsheets are not large, and may not be representative of a larger class of spreadsheets. The fact that the faults placed in the spreadsheets were synthetic (seeded) may also affect our ability to generalize results pertaining to the efficacies of our test suites.

Our test cases are constructed to cover the du-associations in the original spreadsheets; the faulty versions may contain different du-associations, and thus, test suites that are du-adequate for the original spreadsheets may not be adequate for the faulty versions. Perhaps of greatest significance,

our du-adequate test suites may not be representative of those that would be constructed by typical users of our methodology. Random selection of test suites from pools ensured that we studied a range of du-adequate test suites, but we cannot claim that the composition of those individual test suites is necessarily representative of suites that would be created by users of our methodology. Nor can we claim that the randomly generated suites represent test suites that users not in possession of our methodology would create: informed "ad hoc" test design by users may improve test suite efficacy even in the absence of a testing environment supporting a more rigorous methodology.

In general, however, threats involving subject representativeness can be addressed only by additional studies on additional subjects.

5. CONCLUSION

Due to the popularity of commercial spreadsheets, spreadsheet languages are being used to produce software that influences important decisions. Further, due to recent advances from the research community that expand its capabilities, the use of this paradigm is likely to continue to grow. We believe that the fact that such a widely used and growing class of software often has faults should not be taken lightly.

To address this issue, we have developed a methodology that brings some of the benefits of formal testing to this class of software. Key to its appropriateness for the spreadsheet paradigm are four features. First, our methodology accommodates the dependence-driven evaluation model, and is compatible with evaluation engine optimizations, such as varying evaluation orders and value-caching schemes. Second, our collection of algorithms is structured such that their work can be performed incrementally, and hence can be tightly integrated with the highly interactive environments that characterize spreadsheet programming. Third, our algorithms are reasonably efficient given their context, because the triggers that require immediate response from most of the algorithms also require immediate response to handle display and/or value cache maintenance, and the same data structures must be traversed in both cases. The only algorithm that adds more than a constant factor is `Validate`, whose cost is the same order as the cost of recalculating the cell being validated. Finally, our methodology does not require knowledge of testing theory; instead, our algorithms track the "testedness" of the spreadsheet incrementally, and use visual devices to call attention to insufficiently tested interactions.

The methodology presented in this article addresses only one of the important problems in dealing with spreadsheet errors. Other testing problems have been examined in the context of the imperative language paradigm, including the problems of generating test inputs, validating test outputs, automating the replay of tests for regression testing, detecting nonexecutable code, and integrating testing and debugging. These problems are also important in the context of the spreadsheet language paradigm, and in our ongoing work we are investigating them. We are also

working on a way to scale up the methodology by taking into account blocks of cells whose formulas are shared or copies of one other, and appropriately sharing information about their testedness [Burnett et al. 1999].

Our empirical results suggest that our methodology can achieve fault detection results comparable to those achieved by analogous techniques for testing imperative programs. These results are important, because they imply that the potential benefit of this approach to spreadsheet users may be substantial. However, these results do not address the ability of users to utilize our methodology. Thus, we are continuing our work on the implementation of our methodology, and it will soon be robust enough to allow a series of user studies.

We intend to examine the relative effectiveness of our approach among different user groups. Although a large proportion of the users creating spreadsheets are nonprogrammers, a significant number of complex spreadsheets are also created by programmers, including some that are sold or otherwise disseminated as templates. We might expect users that have programming experience to perform differently in testing tasks than users who do not have such experience. We hope ultimately to provide effective testing methodologies to both of these user populations.

APPENDIX

CRG CONSTRUCTION FOR CONDITIONAL SUBEXPRESSIONS

One form of expression apparent in the grammar shown in Figure 2 has not been treated in a manner suitable for the needs of end-users by the literature on imperative programs, and merits special attention.

In spreadsheet languages, because there are only expressions and hence no statements, the `if` construct instantiates an expression instead of a statement, which allows "conditional subexpressions," i.e., `if` expressions included as subexpressions in other expressions. For example, a cell *C*'s formula can consist of the expression `x + (if y=0 then 1 else 2)`. But what is the proper formula graph for such an expression? The approach of Aho et al. [1986] parses this expression into a pair of intermediate code statements (instead of expressions): (i) `if y=0 then tmp=1 else tmp=2` and (ii) `C = x + tmp`. A formula graph for this pair of statements consists of the nodes required to represent (i), followed by the node required to represent (ii), and these nodes include constructs not legal in formulas to represent assignment, variables, and statements. A disadvantage of this approach, especially to an audience of end-users, is that if reasoning about the testedness of a cell or relationship is based in part upon constructs not allowed in spreadsheet formulas, feedback about the testedness of a particular cell or relationship might not be understandable to the user.

A second approach to modeling this expression is to use a "virtual cell" `tmp` to represent the computation of `if y=0 then 1 else 2` and treat *C*'s formula as consisting only of `x + tmp`. However, since this approach introduces a cell not actually present in the spreadsheet, feedback based on reasoning about `tmp` may still not be understandable to the user.

A third approach is to distribute operand `x` over the `if` expression, obtaining `if y=0 then x+1 else x+2` (in the formula graph, not in the user view), and model this like any other `if` expression. We have selected the third approach, because we believe it to be the most likely to be understandable by end-users. Although we do not expect conditional subexpressions to be used widely by spreadsheet users, we also note that they are supported by popular spreadsheet languages, and hence need to be supported by our spreadsheet testing methodology.

ACKNOWLEDGMENTS

We thank the Visual Programming Research Group for their work on the Forms/3 implementation and for their feedback on the testing methodology. Thanks especially to Anurag Agrawal, Joseph Davis, Rebecca Walpole Djang, David Haney, and Virginia Perkins for their contribution of faulty programs. We thank Roland Untch for his assistance with the statistical analysis of our empirical data. We also thank the reviewers of this article, and of the ICSE version of portions of this article, for comments that materially improved both the work and the article.

REFERENCES

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

AMBLER, A. L., BURNETT, M. M., AND ZIMMERMAN, B. A. 1992. Operational versus definitional: A perspective on programming paradigm. *IEEE Computer 25*, 9 (Sept.), 28–43.

AZEM, A., BELLI, F., JACK, O., AND JEDRZEJOWICZ, P. 1993. Testing and reliability of logic programs. In *Proceedings of the Fourth International Symposium on Software Reliability Engineering*. 318–327.

BELLI, F. AND JACK, O. 1995. A test coverage notion for logic programming. In *Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering* (ISSRE '95, Toulouse, France). IEEE Computer Society, Washington, DC, 133–142.

BROWN, P. AND GOULD, J. 1987. Experimental study of people creating spreadsheets. *ACM Trans. Off. Inf. Syst. 5*, 3 (July), 258–272.

BURNETT, M. M. AND GOTTFRIED, H. J. 1998. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Trans. Comput. Hum. Interact. 5*, 1 (Mar.), 1–33.

BURNETT, M., ATWOOD, J., AND WELCH, Z. 1998. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings of the IEEE Symposium on Visual Languages* (Sept.).

BURNETT, M., HOSSLI, R., PULLIAM, T., VANVOORST, B., AND YANG, X. 1994. Toward visual programming languages for steering in scientific visualization: A taxonomy. *IEEE Comput. Sci. Eng. 1*, 4, 44–62.

BURNETT, M., SHERETOV, A., AND ROTHERMEL, G. 1999. Scaling up a "What you see is what you test" methodology to spreadsheet grids. In *Proceedings of the IEEE Symposium on Visual Languages* (Sept.). 30–37.

CHI, E. H.-H., KONSTAN, J., BARRY, P., AND RIEDL, J. 1997. A spreadsheet approach to information visualization. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology* (UIST '97, Banff, Alberta, Canada, Oct. 14–17), G. Robertson and C. Schmandt, Chairs. ACM Press, New York, NY, 79–80.

CHRIST, R. 1975. Review and analysis of color coding research for visual displays. *Hum. Factors 17*, 6, 542–570.

CLARKE, L. A. 1976. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng. SE-2*, 3, 215–222.

CLARKE, L. A., PODGURSKI, A., RICHARDSON, D. J., AND ZEIL, S. J. 1989. A formal evaluation of data flow path selection criteria. *IEEE Trans. Softw. Eng. 15*, 11 (Nov.), 1318–1332.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. 1992. Rigorous data flow testing through output influences. In *Proceedings of the 2nd Symposium on Irvine Software* (Mar.).

DUPUIS, C. AND BURNETT, M. 1997. An animated turing machine simulator in Forms/3. i97-60-08 (July).

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July), 319–349.

FRANKL, P. AND WEISS, S. 1993. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Softw. Eng. 19*, 8 (Aug.), 774–787.

FRANKL, P. G. AND WEYUKER, E. J. 1988. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng. 14*, 10 (Oct.), 1483–1498.

GREN, T. R. G. AND PETRE, M. 1996. Usability analysis of visual programming environments: A "cognitive dimensions" framework. *J. Visual Lang. Comput. 7*, 2, 131–174.

GUPTA, R., HARROLD, M. J., AND SOFFA, M. L. 1996. Program slicing-based regression testing techniques. *J. Softw. Test. Verific. Reliab. 6*, 2 (June), 83–112.

HARROLD, M. J. AND SOFFA, M. L. 1988. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance* (Oct.). 362–367.

HUDSON, S. E. 1991. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Trans. Program. Lang. Syst. 13*, 3 (July), 315–341.

HUTCHINS, M., FOSTER, H., GORADIA, T., AND OSTRAND, T. 1994. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering* (ICSE '94, Sorrento, Italy, May 16–21), B. Fadini, L. Osterweil, and A. van Lamsweerde, Chairs. IEEE Computer Society Press, Los Alamitos, CA, 191–200.

JOHNSON, R. 1992. *Elementary Statistics*. 6th. Duxbury Press, Boston, MA.

KUHN, W. AND FRANK, A. U. 1997. The use of functional programming in the specification and testing process. In *Proceedings of the International Conference and Workshop on Interoperating Geographic Information Systems* (Dec.).

LANDI, W. 1992. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. 1*, 4 (Dec.), 323–337.

LANDI, W. AND RYDER, B. G. 1991. Pointer-induced aliasing: A problem taxonomy. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages* (POPL '91, Orlando, FL, Jan. 21–23), D. Wise, Chair. ACM Press, New York, NY, 93–103.

LANDI, W. AND RYDER, B. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation* (SIGPLAN '92, San Francisco, CA, June 17–19), R. L. Wexelblat, Ed. ACM Press, New York, NY, 235–248.

LASKI, J. AND KOREL, B. 1983. A data flow oriented program testing strategy. *IEEE Trans. Softw. Eng. 9*, 3 (May), 347–354.

LEOPOLD, J. AND AMBLER, A. 1997. Keyboardless visual programming using voice, handwriting, and gesture. In *Proceedings of the IEEE Symposium on Visual Languages* (VL97, Capri, Italy, Sept.). IEEE Computer Society Press, Los Alamitos, CA, 28–35.

LUO, G., BOCHMANN, G., SARIKAYA, B., AND BOYER, M. 1992. Control-flow based testing of Prolog programs. In *Proceedings of the 3rd International Symposium on Software Reliability Engineering*. 104–113.

MARLOWE, T. AND RYDER, B. 1990. An efficient hybrid algorithm for incremental data flow analysis. *Prin. Prog. Lang.* (Jan.), 184–196.

MURCH, G. M. 1984. Physiological principles for the effective use of color. *IEEE Comput. Graph. Appl. 4*, 11 (Nov.), 49–54.

MYERS, B. A. 1991. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of the Conference on Human Factors in Computing Systems: Reaching through Technology* (CHI '91, New Orleans, LA, Apr. 27–May 2), S. P. Robertson, G. M. Olson, and J. S. Olson, Eds. ACM Press, New York, NY, 243–249.

NTAFOS, S. C. 1984. On required element testing. *IEEE Trans. Softw. Eng. 10*, 6 (Nov.), 132–139.

OFFUTT, A. J., PAN, J., TEWARY, K., AND ZHANG, T. 1996. An experimental evaluation of data flow and mutation testing. *Softw. Pract. Exper. 26*, 2, 165–176.

OUABDESSELAM, F. AND PARISSIS, I. 1995. Testing techniques for data-flow synchronous programs. In *Proceedings of the Second International Workshop on Automated and Algorithmic Debugging* (AADEBUG'95, May).

PANKO, R. AND HALVERSON, R. 1996. Spreadsheets on trial: A survey of research on spreadsheet risks. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences* (ICSS '96, Maui, Hawaii, Jan.). IEEE Computer Society Press, Los Alamitos, CA.

PERRY, D. E. AND KAISER, G. E. 1990. Adequate testing and object-oriented programming. *J. Object Oriented Program. 2*, 5 (Jan./Feb.), 13–19.

POLLOCK, L. L. AND SOFFA, M. L. 1989. An incremental version of iteractive data flow analysis. *IEEE Trans. Softw. Eng. 15*, 12 (Dec.), 1537–1549.

RAPPS, S. AND WEYUKER, E. J. 1985. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr.), 367–375.

REICHWEIN, J., ROTHERMEL, G., AND BURNETT, M. 1999. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. In *Proceedings of the 2nd Conference on Domain Specific Languages* (Oct.). 25–38.

ROTHERMEL, G. AND HARROLD, M. J. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol. 6*, 2, 173–210.

ROTHERMEL, G. AND HARROLD, M. J. 1994. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis* (ISSTA '94, Seattle, WA, Aug. 17–19), T. Ostrand, Ed. ACM Press, New York, NY, 169–184.

ROTHERMEL, G., LI, L., AND BURNETT, M. 1997. Testing strategies for form-based visual programs. In *Proceedings of the 8th International Symposium on Software Reliability Engineering* (ISSRE 97, Nov.). 96–107.

ROTHERMEL, G., LI, L., DUPUIS, C., AND BURNETT, M. 1998. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 20th International Conference on Software Engineering* (ICSE '98, Kyoto, Japan, Apr.). IEEE Press, Piscataway, NJ, 198–207.

SHNEIDERMAN, B. 1998. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 3rd. Addison-Wesley, Reading, MA.

SMEDLEY, T., COX, P., AND BYRNE, S. 1996. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *Proceedings of the Conference on Advanced Visual Interfaces* (AVI'96, Gubbio, Italy, May), T. Catarci, M. F. Costabile, S. Levialdi, and G. Santucci, Eds. ACM Press, New York, NY.

VIEHSTAEDT, G. AND AMBLER, A. 1992. Visual representation and manipulation of matrices. *J. Visual Lang. Comput. 3*, 3 (Sept.), 273–298.

WEYUKER, E. J. 1982. On testing non-testable programs. *Computer J. 15*, 4, 465–470.

WEYUKER, E J. 1986. Axiomatizing software test data adequacy. *IEEE Trans. Softw. Eng. SE-12*, 12 (Dec.), 1128–1138.

WEYUKER, E. J. 1993. More experience with dataflow testing. *IEEE Trans. Softw. Eng. 19*, 9 (Sept.), 912–919.

WILCOX, E. M., ATWOOD, J. W., BURNETT, M. M., CADIZ, J. J., AND COOK, C. R. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI '97, Atlanta, GA, Mar. 22–27), S. Pemberton, Ed. ACM Press, New York, NY, 258–265.

WONG, W. E., HORGAN, J. R., LONDON, S., AND MATHUR, A. P. 1995. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering* (ICSE-17, Seattle, WA, Apr. 23–30), D. Perry, Chair. ACM Press, New York, NY, 41–50.

YANG, S., BURNETT, M., DEKOVEN, E., AND ZLOOF, M. 1997. Representation design benchmarks: A design-time aid for VPL navigable static representations. *J. Visual Lang. Comput. 8*, 5/6 (Oct/Dec), 563–599.