Testing Strategies for Form-Based Visual Programs

Gregg Rothermel
Department of
Computer Science
Oregon State University
Corvallis, OR 97331
grother@cs.orst.edu

Lixin Li
Department of
Computer Science
Oregon State University
Corvallis, OR 97331
lili@cs.orst.edu

Margaret Burnett
Department of
Computer Science
Oregon State University
Corvallis, OR 97331
burnett@cs.orst.edu

Abstract

Form-based visual programming languages, which include electronic spreadsheets and a variety of research systems, have had a substantial impact on end-user computing. Research shows that form-based visual programs often contain faults, and that their creators often have unwarranted confidence in the reliability of their programs. Despite this evidence, we find no discussion in the research literature of techniques for testing or assessing the reliability of form-based visual programs. This paper addresses this lack. We describe differences between the form-based and imperative programming paradigms, and discuss effects these differences have on strategies for testing form-based programs. We then present several test adequacy criteria for form-based programs, and illustrate their application. We show that an analogue to the traditional "all-uses" dataflow test adequacy criterion is well suited for code-based testing of form-based visual programs: it provides important error-detection ability, and can be applied more easily to form-based programs than to imperative programs.

1. Introduction

Form-based visual programming languages provide a declarative approach to programming, characterized by a dependency-driven, direct-manipulation working model. Users of form-based languages create or manipulate cells, and define formulas for those cells. These formulas reference values contained in other cells and use them in calculations. Whenever a cell's formula is defined, the underlying evaluation engine calculates the cell's value, recalculates the values of cells that reference recalculated cells, and displays new results on the screen.

Form-based visual programming languages include, as a subclass, electronic spreadsheet systems. These systems are among the most widely used types of software; a large population of end-users employ them to perform a variety of computational tasks. The form-based visual language paradigm is also employed in numerous research tools, which seek to extend its power and applicability. For example, there is research into using form-based languages for database access [34], for providing steerable simulation environments for scientists [8], and for supporting the specification and implementation of full-featured GUIs [20].

Despite the end-user appeal of form-based languages, and the perceived simplicity of the paradigm in comparison to the imperative-language paradigm, research shows that form-based visual programs often contain faults. For example, in one empirical study of experienced spreadsheet users [6], 44 percent of the spreadsheets created by those users were found to contain user-generated faults, a rate comparable to that estimated in the trade press [9]. Compounding this problem, creators of spreadsheets express unwarranted confidence in the reliability of their programs [6].

In spite of this evidence, we find no discussion in the research literature of techniques for testing or assessing the reliability of form-based visual programs. In fact, most research on program testing to date (e.g. [11, 13, 17, 25, 33]) is directed at imperative programs. Some recent research [4, 5, 19] considers problems of testing and reliability determination for logic programs written in Prolog. Although the logic paradigm is like the form-based 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 form-based paradigm that the testing techniques developed for Prolog cannot be applied to the form-based paradigm.

In this paper, we discuss strategies for testing form-based visual programs. We show that significant differences exist between form-based and imperative programs, and that these differences have implications for testing methodologies. Form-based programs require testing methodologies that support an incremental input and validation process. Furthermore, because form-based programs are dependence driven, they are more appropriately tested using adequacy criteria that are not control-flow-based. Finally, the interactive nature of form-based programs provides opportunities for integrating testing, debugging, and validation activities to an extent that is not found with standard imperative languages.

Focusing on code-based testing, we next present several test adequacy criteria for form-based programs, and illustrate their application. We first define and illustrate the use of criteria that focus on formulas, and a criterion that focuses on simple dependencies between cells. We show that these criteria can fail to exercise important interactions within form-based programs, and in doing so, miss a class of faults that are prominent in those programs. We then define an analogue to the traditional "all-uses" dataflow test adequacy criterion, that provides more effective test coverage for form-based programs, because it exercises interactions both between and within cells.

2. Form-based visual languages

In form-based languages, users set up forms and specify their contents in order to program. Some of these contents are left blank, with the expectation that they will be filled in later. To "run" these programs, users fill in these blanks on the forms. The contents of a form are a collection of cells; each cell's value is defined by that cell's formula. Cells left blank simply have no formulas, users fill in those blanks by entering (constant) formulas. The best-known examples of form-based languages are commercial spreadsheets (see Figure 1), but there are also many research systems based upon this paradigm. In its pure form, a form-based program follows Alan Kay's "value rule", which states that a cell's value is defined solely by the formula explicitly specified for that cell by the user [16]. Spreadsheets without macros exemplify "pure" form-based programming.

Examples of research form-based languages include Action Graphics [14], a form-based language for graphics animations; NoPumpG [18] and its successor NoPump2 [31], which are form-based languages designed to support interactive graphics; a system described by Smedley et al.[27], which incorporates the visual dataflow language Prograph into a spreadsheet; Wilde's WYSIWYC spreadsheet [32], a project aimed at improving spreadsheets by making the visible structure of the spreadsheet more closely match computational structures; and C32 [20], a form-based system that uses graphical techniques along with inference to specify constraints in user interfaces.

In this paper, we present examples of form-based programs in the research language Forms/3 [3, 7]. As in other form-based languages, to program in Forms/3, the user creates cells and gives each cell a formula; as soon as the user enters a formula, it is evaluated and the result is displayed.

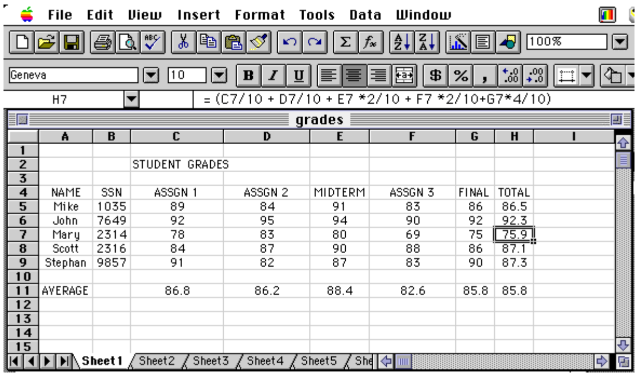

Figure 1. Spreadsheet to calculate student grades. Each row shows a student's name, SSN, grades, and total score. The last row averages grades and scores. In this spreadsheet, cells that show totals and averages contain formulas. The formula for cell H7, $(C7/10+D7/10+E7*2/10+F7*2/10+G7*4/10)$**, is displayed near the top of the spreadsheet. Other cells contain text or constants.**

Figures 2 and 3 show how a programmer would construct a single digit LED in Forms/3. Figure 2 shows each cell with its formula displayed. There are seven cells that make up the LED (one for each line) and one input radio button set. Each cell checks to see if a particular horizontal or vertical line is required for the selected input.

After all the programming is finished, the formula tabs and borders can be hidden to clean things up. The horizontal and vertical cells can also be hidden to reach a user view, as shown in Figure 3. Now a user simply has to click on the different buttons of the input set and the selected digit will be displayed in the LED.

2.1. A simple form-based visual language.

For this paper, we have selected a subset of Forms/3 that is representative of both commercial and research form-based visual languages. The subset has ordinary spreadsheet-like formulas for mathematics and conditional operations, and includes support for elementary graphics using the same mechanism. Figure 2 was programmed using this subset. A program in this language is entirely determined by the collection of cell formulas. The grammar for the formulas in this subset is shown in Table 1. Type issues are ignored in this grammar because most form-based languages are dynamically typed; therefore, the above syntax is accepted regardless of types. Formulas whose types do not "work out" at runtime evaluate to a distinguished error value.

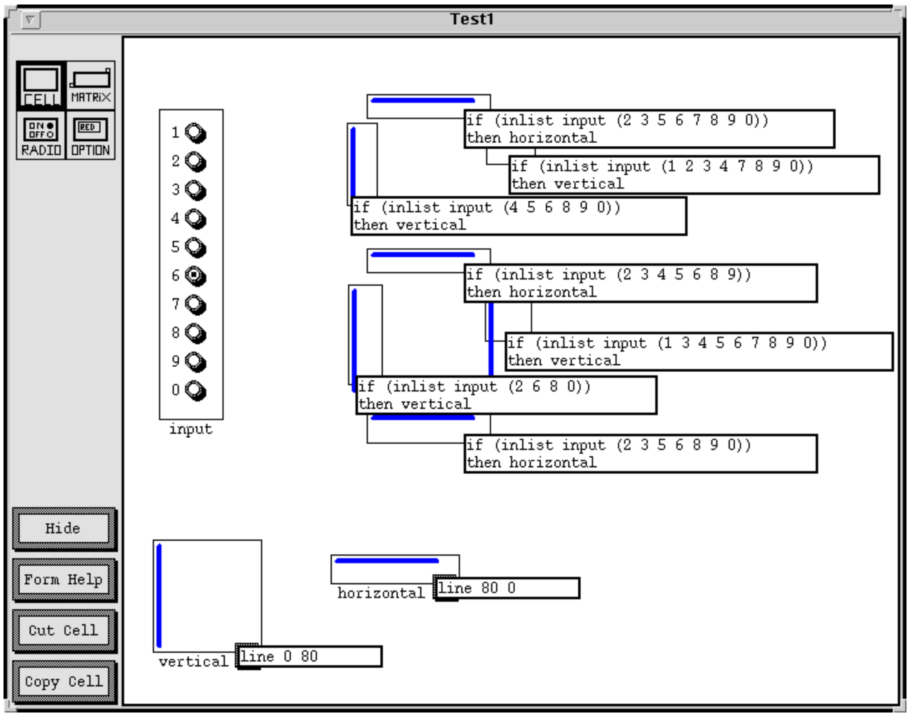

Figure 2. Programming a single digit LED in Forms/3.

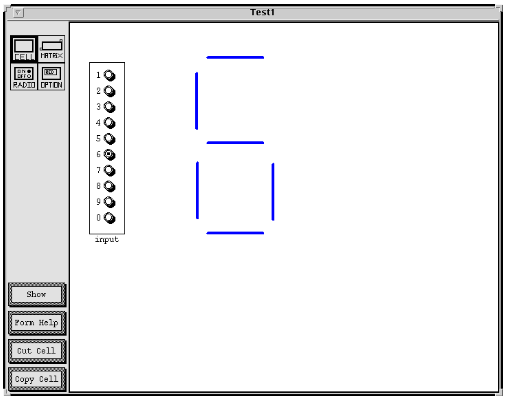

Figure 3. A single digit LED output.

formula ::= *BLANK* | expr
expr ::= *CONSTANT* | *CELLREF* | infixExpr |
prefixExpr | ifExpr
infixExpr ::= subExpr infixOperator subExpr
prefixExpr ::= unaryPrefixOperator subExpr |
binaryPrefixOperator subExpr subExpr
ifExpr ::= IF subExpr THEN subExpr ELSE subExpr |
IF subExpr THEN subExpr
subExpr ::= *CONSTANT* | *CELLREF* | (expr)
infixOperator ::= $+ \mid - \mid * \mid / \mid$ AND | OR | = | ...
unaryPrefixOperator ::= NOT | ...
binaryPrefixOperator ::= LINE | BOX | INLIST | ...

Table 1. Grammar for formulas

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 "elseless" 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 multi-token subexpressions are always wrapped in parentheses.

From this grammar, it is clear 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. This fact has implications for testing, as we show in Section 3.

Because the language of formulas does not distinguish among input, output, and intermediate calculations, it is easy for a user to accidentally replace an interim calculation with an input, to forget to change previously-established (default) input formulas, and so on. To alleviate this, most form-based visual languages include the ability to "lock down" some of the cells to prevent accidental modification of the formulas. When present, this feature can be used by a testing mechanism to differentiate inputs from calculations.

2.2. Evaluation strategies for form-based languages

In essence, the evaluation strategies used in form-based languages follow the principles of either eager evaluation or lazy evaluation, although a variety of optimizations and combinations are employed by some languages. 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, 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 determined dynamically, so this is not as conservative as the eager approach. 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. Because form-based languages are visual, keeping many cells on display automatically, at least some of the cell values are usually saved. This means that a lazy evaluation engine also needs to keep track of which saved values are up-to-date if the user has started changing formulas. There are several methods for doing so, but their mechanism is not relevant to the issues in this paper, and we assume for simplicity of exposition that a cell value has never been computed before. It has been shown that eager evaluation produces the same answers as lazy evaluation, provided that both terminate. However, lazy evaluation computes fewer cells.

In form-based visual languages, some cells will be on the screen and 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 scroll or otherwise move cells on and off the screen through direct manipulation. Which cells are on-screen determines which input cells will be noticed and attended to, and which output cells will be seen. In the case of languages following lazy evaluation, it also determines which cells will be computed, since lazy evaluation is output-driven.

3. Issues and opportunities

In this section we describe several differences between imperative and form-based programs, and discuss effects these differences have on strategies for testing form-based programs. The comparison reveals several opportunities for utilizing characteristics of form-based languages in testing.

3.1. Types of faults found in form-based programs

Empirical data [6] suggests that most faults that occur in form-based programs involve incorrect formulas. Some faults, such as syntax errors, cycles in dependencies, and references to non-existent cells, can be detected by the programming environment at the time the program is created; these do not concern us here. Of the faults that cannot be thus detected, most involve incorrect or missing references to cells. A smaller class of faults involve the erroneous use of operators or constants. Faults may occur in computational expressions, directly affecting the values that are assigned to cells, or in predicate expressions, directly affecting the flow of control within cell formulas and indirectly affecting assignments of values to cells.

3.2. Specification-based versus code-based testing

Whereas imperative languages have been intended for use primarily by professional programmers, form-based languages are intended for use by a variety of audiences, ranging from professional programmers to end users. Few of these users are likely to create specifications for their programs. Moreover, even when specifications exist, evidence suggests that code-based testing techniques provide an effective mechanism for detecting faults [15, 33] and increasing software reliability [10]. Thus, we focus on techniques for code-based testing of form-based programs. In the absence of specifications, we assume that the programmer/tester serves as the *oracle* to validate the correctness of test outputs. Some consequences of this assumption are discussed in [28].

3.3. Incremental versus nonincremental testing

We can think of a form-based program as a function that maps a vector of "input cells" to a vector of "output cells." D, the domain of this function, is a set of vectors whose elements consist of values for their corresponding input cells. A test t for a form-based program is an element of D. In this paradigm, t is an assignment of values to all input cells.

With imperative programs, tests are often run in three steps: first, inputs are initialized; next, the program is executed; and finally, results are validated. This process is repeated for each test in the test suite. A similar process applies to form-based programs: to run a test we initialize all input cells; then, we "execute" the program; finally, we validate results. With form-based programs, however, program execution is accomplished automatically by the evaluation engine, and this engine updates output cells following *each* input, not just the final one. This immediate feedback for user actions is called *responsiveness*, and is a common characteristic of the form-based visual language paradigm [2]. Responsiveness in form-based programs promotes the use of an *incremental testing process*: given a form-based program in which input cells contain values, we incrementally alter those values; following each alteration we validate the contents of relevant output cells.

Incremental testing can reduce the effort required to test form-based programs, by reducing the number of inputs that the tester must enter. Furthermore, when we apply subsequent tests incrementally, we need only validate outputs that are affected by modified inputs; outputs that depend only on inputs entered previously do not require revalidation. For programs that contain large numbers of cells, the reduction in effort can be substantial. Thus, a testing methodology for form-based program should accommodate the use of an incremental testing process.

3.4. Applicability of criteria

Most code-based test adequacy criteria for imperative programs are defined in terms of control flow. For example, data-flow and basis path adequacy criteria are defined in terms of paths through control flow graphs [11, 23]. Such definitions suffice for imperative programs, in which flow of control between code constructs is explicit at the language level, and program execution follows control flow; however, the situation is different for form-based programs. At the language level, in form-based programs, execution follows explicit control flow only within formulas, and the evaluation order of cells is dependence-driven. At the implementation level, however, a particular evaluation engine executes a program according to a particular evaluation scheme, inducing a specific evaluation order on cells.

This difference between language level and implementation level views has consequences for code-based test adequacy criteria. For example, consider the basis path criterion, which requires coverage of all linearly independent paths through a program. Given form-based program P, we can determine a finite, maximal set of cell orderings that each yield a correct evaluation of P, given its dependencies. Each ordering represents a path through P that joins cell formulas to one another; the set of orderings constitutes the set of all linearly independent paths through P. An arbitrary evaluation engine could use any of these orderings to evaluate P. Under a particular engine, however, only one ordering is dynamically executed; for that engine, only that ordering corresponds to an executable path.

A test adequacy criterion is *applicable* if, for every program P, there exists a finite test set that is adequate according to that criterion for P [29]. One of the ways in which a test adequacy criterion fails to be applicable is if it requires coverage of *nonexecutable* code components. Because adequacy criteria often involve code components that can be nonexecutable, a common tactic is to render the criteria applicable by redefining them, such that they require coverage only of executable components [11]. When this tactic is employed, testers must determine whether code components are executable, in order to complete their evaluations of test adequacy. Determining whether a component is executable is a difficult problem [30]; adequacy criteria that identify fewer nonexecutable components may, for this reason, be preferable to those that identify more.

Applied to form-based programs at the language level, control-flow-based adequacy criteria that require coverage of all valid cell evaluation orders identify components that are predominantly nonexecutable under a particular evaluation engine. Such criteria are not applicable; restricting them to executable components will require inordinate effort on the parts of testers.

An alternative control-flow-based adequacy criterion, applied at the implementation level, could focus on cell evaluation orders that may be exercised by a particular evaluation engine. Essentially, this approach requires definitions of adequacy criteria at the implementation level, rather than the language level. This strategy has the advantage of closely relating program coverage to program execution; however, it also has drawbacks. Building engine evaluation order into a testing criterion presents the form-based program tester with a view of the program that is not natural to the form-based language paradigm. A programmer does not and should not need to know, for a given form-based program, the order in which cells are evaluated, except where that order relates to dependencies. Furthermore, to implement adequacy criteria that depend on engine evaluation order, we must build testing tools that are cognizant of an underlying engine's evaluation order; this requirement reduces the generality of the tools.

In this work, we focus on engine-independent adequacy criteria; that is, criteria that can be satisfied under any valid evaluation engine. Under this approach, to avoid problems with applicability, we require criteria that can be defined and measured independent of cell evaluation order.

3.5. Integrated development, testing and debugging

Form-based programs are responsive. When a programmer modifies a form-based program, the modified code is immediately executed, and affected results are immediately displayed. There is no separate compile-and-link process to be completed before outputs can be displayed and validated. As a result, the processes of creating, testing, and debugging form-based programs are tightly coupled.

An environment for developing form-based programs can capitalize on this coupling. Form-based language programmers can incrementally test and validate modifications more readily than can imperative language programmers. When a test results in a failure, the same data that lets testing utilities in the programming environment track and indicate tested components can be used to identify components that dynamically affect the execution, and help the programmer locate the fault. When the programmer corrects the fault, the testing utilities can immediately calculate code components affected by the modification that should now be revalidated. These components can immediately be retested.

3.6. The user interface

Although a few form-based language programmers will understand concepts such as test adequacy, many others will be able to take advantage of more rigorous testing methodologies only if those methodologies are appropriately presented. In this paper, we focus on providing theoretically sound strategies for testing form-based visual programs. Ultimately, for these strategies to be useful to end-user programmers, they must be presented in an intuitive manner.

One concept that is not foreign to form-based language programmers is the concept of cells interacting through cell references [6]. Testing methodologies framed in terms of cell interactions should be accessible to such programmers. We can further support their understanding by presenting testing-related information in a manner that is integrated with the form-based language paradigm; that is, by providing a testing environment in which that information is presented visually.

4. Adequacy criteria for form-based programs

In this section we describe several code-based test adequacy criteria for use with form-based visual programs. To simplify the presentation, we make several assumptions about the programs that are to be tested. We assume that cells can be partitioned into *input cells* and *non-input cells* – the former include the cells that constitute the input vector to the program, and the latter include the cells whose output values may require validation. We also assume that formulas for input cells do not contain references to other cells, that cell references can be resolved statically, that each predicate expression contains at least one cell reference, that formulas for non-input cells are locked, that recursion is not allowed, and that formulas are not entered as test inputs.

As a running example, we use the Forms/3 program `rootsolver`, displayed in Figure 4, that calculates the solutions of the equations $ax^2 + bx + c = 0$ or $bx + c = 0$. In the figure, cell formulas are expanded; each cell's value is shown at the top-left corner of the cell, and each cell's formula is shown in the box at bottom-right corner of the cell. Cells a, b, and c are input cells for coefficients a, b, and c, respectively; cells f and g display the roots calculated for the equation given those coefficients; and cells d and e are intermediate cells used in the calculation. In the figure, the cells display values for the case where the input to the program is {a=1, b=-3, c=2}. When $a = 0$ the program calculates the root of $bx + c = 0$.

4.1. An abstract model for form-based programs

Code-based test adequacy criteria are frequently defined on abstract models of programs rather than directly on code itself; we use this approach. Given form-based program P, we construct a *cell relation graph* (CRG) that models two properties of P: flow of control within P's formulas, and dependencies between P's cells. To model the flow of control within cell formulas, we represent each cell by a *formula graph*; these graphs are comparable to the control flow graphs used to represent procedures in imperative programs. A formula graph is a directed graph in which each node represents a simple or conditional expression in a cell formula, and each edge represents the flow of control between expressions. Unique entry and exit nodes represent initiation and termination of the evaluation of the formula. We call the set of formula graphs obtained from a program P the *formula graph set* for P.

Figure 5 displays the formula graph for cell `f` of `rootsolver`. In the figure, expression nodes, shown as ellipses, represent simple expressions. Predicate nodes, shown as rectangles, represent conditional expressions; labeled edges out of these nodes represent control paths taken when the conditional expression evaluates to the value of the edge label. For reference, we label each node in the graph with the name of the associated cell and an integer.

Given cells A and B in program P, if B's formula references A, then B's computation may make use of the value associated with A: in this case we say that B is *cell dependent* on A. To model cell dependencies, we represent them in the CRG as edges between formula graphs: for each pair of cells A and B in program P, if B is cell dependent on A, we add an edge from the exit node A_x of A's cell formula graph to the entry node B_e of B's cell formula graph.

Figure 6 displays the CRG for `rootsolver`. In the figure, formula graphs are enclosed within dotted rectangles, and dashed lines depict cell dependencies. To simplify the presentation, however, we depict cell dependence edges as beginning and terminating at the dotted rectangles.

Cell dependence edges do not represent control flow. If (A_x,B_e) is a cell dependence edge in the CRG for program P, then whether A or B or both must be executed for a given input, or whether execution of B will immediately follow execution of A, depends on the underlying evaluation scheme and on other dependencies in P. In contrast, edges within formula graphs do represent control flow.

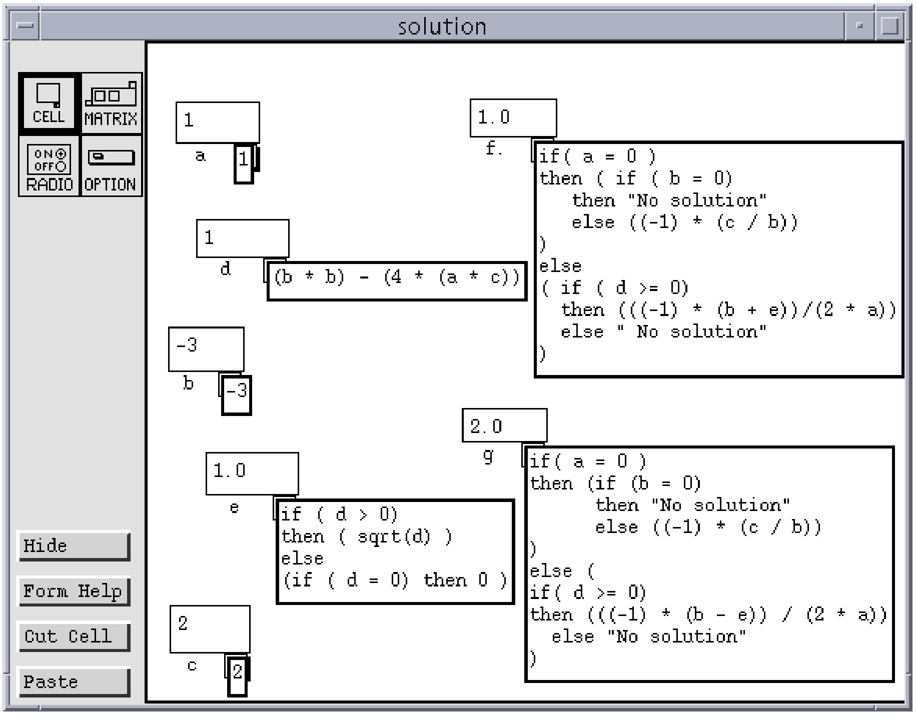

Figure 4. Forms/3 program to obtain the root(s) of quadratic or linear equations.

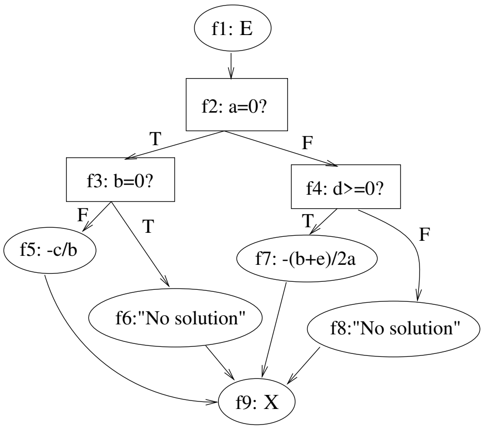

Figure 5. Formula graph for cell `f` of `rootsolver`.

An evaluation of formula F can be said to traverse a path through the formula graph $\bar{F}$ for F, beginning at $\bar{F}$'s entry node and ending at its exit node.

4.2. Node and edge adequacy criteria

For imperative programs, two simple code-based test adequacy criteria are node and edge adequacy criteria, which require selection of test data that exercises every node and edge, respectively, in the control flow graph for a procedure. Because nodes and edges may be unreachable, these criteria are rendered *applicable* by restricting coverage requirements to executable program components [11]. We use formula graph sets to define analogues, for form-based programs, of these applicable node and edge adequacy criteria. Test t exercises a node N in formula graph G if t causes the evaluation of the formula that corresponds to G, and that evaluation traverses a path through G that includes N. A test suite T is *node-adequate* for form-based program P if, for each formula graph G in the formula graph set for P, for each dynamically executable node N in G, there is at least one test in T that exercises N. Similarly, test t exercises an edge $(N1, N2)$ in formula graph G if t causes the evaluation of the formula that corresponds to G, and that evaluation traverses a path through G that includes $(N1, N2)$. A test suite T is *edge-adequate* for form-based program P if, for each formula graph G in the formula graph set for P, for each executable edge $(N1, N2)$ in G, there is at least one test in T that exercises $(N1, N2)$.

Table 2 presents a test suite that is both node- and edge-adequate for `rootsolver`. As the table illustrates, each test case causes multiple cell formulas to be evaluated, and exercises coverable components in each of those cells; however, no single test case exercises all coverable components.

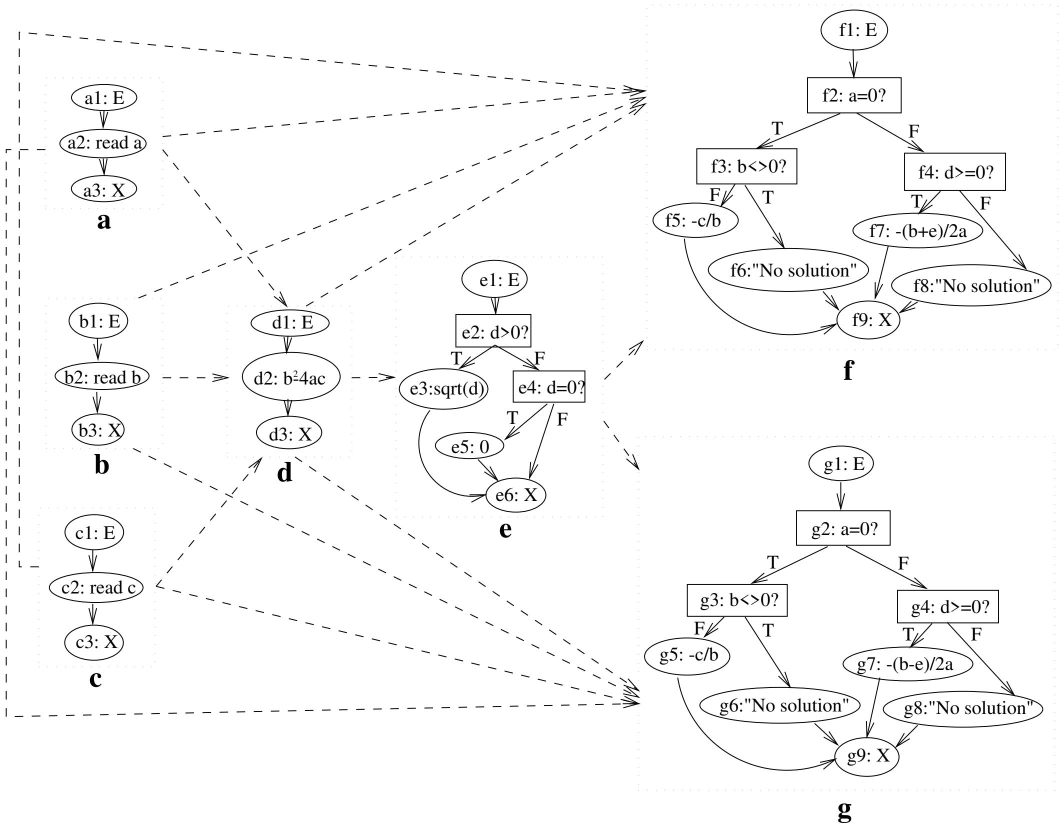

Figure 6. Cell relation graph for `rootsolver`.

For example, test case 1 enters values in all three input cells (a, b, and c), exercising all edges in those cells. These inputs force cell d to be evaluated; that evaluation exercises all edges in d and assigns value `1` to d. Cell e is now executed; because the value of d is positive, this execution exercises edges ($e1$,$e2$), ($e2$,$e3$), and ($e3$,$e6$) and assigns value `1` to e. Cells f and g are now executed, their execution exercises edges ($f1$,$f2$), ($f2$,$f4$), ($f4$,$f7$), and ($f7$,$f9$), and ($g1$,$g2$), ($g2$,$g4$), ($g4$,$g7$), and ($g7$,$g9$), respectively. Row 1 of Table 2 lists the resulting coverage.

The test suite shown in Table 2 is both node- and edge-adequate for `rootsolver`. A test suite may, however, exercise all nodes in a formula graph without exercising all edges. For example, a test suite for `rootsolver` could exercise every node in the formula graph for cell e without exercising edge ($e4$,$e6$).

4.3. The cell dependence adequacy criterion

As the foregoing example shows, the act of exercising nodes and edges also exercises interactions between cells. However, node and edge adequacy criteria do not explicitly require tests that exercise such interactions. Thus, node and edge-adequate test suites may fail to exercise cell interactions that, if exercised, could reveal faults. For example, suppose the programmer of `rootsolver` mistyped the last line of the formula for cell e as "`(if (d=0) then 2)`". In this case, for inputs $\{a = 1, b = 2, c = 1\}$, the program should calculate double roots `-1` and `-1`; instead, it calculates roots `0` and `2`. The node- and edge-adequate test suite shown in Table 2 does not detect this fault.

To specify an adequacy criterion that explicitly requires the testing of cell interactions, an initial approach focuses on cell dependencies. Let A and B be cells in form-based program P, with associated formula graphs $\bar{A}$ and $\bar{B}$, respectively, and let B be cell dependent on A, represented as edge ($\bar{A}_x$,$\bar{B}_e$) in the CRG for P. Test t *exercises cell dependence edge* ($\bar{A}_X$,$\bar{B}_E$) if t causes evaluation of $\bar{B}$, and that evaluation traverses a path through $\bar{B}$ that contains a node whose associated expression references A. We say that test suite T is *cell-dependence-adequate* for form-based program P if, for each cell dependence edge (X, Y) in the CRG for P, there is some test t in T that exercises (X, Y).

Note that to exercise cell dependence edge (A, B), t need not first cause evaluation of A, followed by evaluation of B. Instead, the edge is exercised if A's value is used during the evaluation of B's formula. This definition supports an incremental testing process: a single input can exercise a

Test	Inputs	Node coverage	Edge coverage
1	$\{$a=1, b=-3, c= 2$\}$	a1,a2,a3,b1,b2,b3,c1,c2,c3 d1,d2,d3,e1,e2,e3,e6 f1,f2,f4,f7,f9,g1,g2,g4,g7,g9	(a1,a2),(a2,a3),(b1,b2),(b2,b3),(c1,c2),(c2,c3) (d1,d2),(d2,d3),(e1,e2),(e2,e3),(e3,e6) (f1,f2),(f2,f4),(f4,f7),(f7,f9),(g1,g2),(g2,g4),(g4,g7),(g7,g9)
2	$\{$a=0, b=3, c=3$\}$	a1,a2,a3,b1,b2,b3,c1,c2,c3 d1,d2,d3,e1,e2,e3,e6 f1,f2,f3,f5,f9,g1,g2,g3,g5,g9	(a1,a2),(a2,a3),(b1,b2),(b2,b3),(c1,c2),(c2,c3) (d1,d2),(d2,d3),(e1,e2),(e2,e3),(e3,e6) (f1,f2),(f2,f3),(f3,f5),(f5,f9),(g1,g2),(g2,g3),(g3,g5),(g5,g9)
3	$\{$a=0, b=0, c=3$\}$	a1,a2,a3,b1,b2,b3,c1,c2,c3 d1,d2,d3,e1,e2,e4,e5,e6 f1,f2,f3,f6,f9,g1,g2,g3,g6,g9	(a1,a2),(a2,a3),(b1,b2),(b2,b3),(c1,c2),(c2,c3) (d1,d2),(d2,d3),(e1,e2),(e2,e4),(e4,e5),(e5,e6) (f1,f2),(f2,f3),(f3,f6),(f6,f9),(g1,g2),(g2,g3),(g3,g6),(g6,g9)
4	$\{$a=1, b=1, c=2$\}$	a1,a2,a3,b1,b2,b3,c1,c2,c3 d1,d2,d3,e1,e2,e4,e6 f1,f2,f4,f8,f9,g1,g2,g4,g8,g9	(a1,a2),(a2,a3),(b1,b2),(b2,b3),(c1,c2),(c2,c3) (d1,d2),(d2,d3),(e1,e2),(e2,e4),(e4,e6) (f1,f2),(f2,f4),(f4,f8),(f8,f9),(g1,g2),(g2,g4),(g4,g8),(g8,g9)

Table 2. Node- and edge-adequate test suite for `rootsolver`.

dependence edge whose source lies in a cell executed by an earlier input.

One advantage of the cell dependence adequacy criterion is that evaluation engines for form-based programs often explicitly track information on cell dependencies; this information is available at no additional expense to testing tools that are integrated with such engines. A second advantage is that the criterion does explicitly address interactions between cells. It is easy to show, however, that cell-dependence-adequate test suites may not be node- or edge-adequate. For example, consider the node- and edge-adequate test suite shown in Table 2. When we remove test cases 3 and 4 from that test suite, the resulting test suite, shown with cell dependence coverage information in Table 3, is cell-dependence-adequate, but is not node- or edge-adequate. For example, the tests in the test suite do not exercise nodes $f7$ or $g7$, or edges $(f7, f9)$ or $(g7, g9)$; the test suite cannot detect faults in $f7$ and $g7$.

Test	Inputs	Cell dependence coverage
1	$\{$a=1, b=-3, c=2$\}$	(a,d),(b,d),(c,d),(d,e),(e,f),(e,g), (a,f),(b,f),(d,f),(a,g),(b,g),(d,g)
2	$\{$a=0, b=3, c=3$\}$	(a,d),(b,d),(c,d),(d,e),(a,f),(b,f), (c,f),(a,g),(b,g),(c,g)

Table 3. Cell-dependence-adequate test suite for `rootsolver`.

4.4. Dataflow adequacy criteria

The cell dependence adequacy criterion requires coverage of some dependencies between cells, but not all; moreover, it does not explicitly consider the effects of the control dependencies that are created by predicate expressions. Analogues of the dataflow test adequacy criteria that have been defined for imperative programs [17, 21, 24] provide a more inclusive alternative.

Dataflow test adequacy criteria relate test coverage to interactions between occurrences of variables; dataflow analysis identifies such interactions. For imperative languages, dataflow analysis classifies the occurrences of variables in a program as definitions or uses, depending on whether those occurrences store values in, or fetch values from, memory, respectively.

A similar classification applies to form-based programs. In form-based programs, however, cells serve as variables, and values can be stored in the memory associated with cell C only by expressions in C's formula, in which the cell name C is implicit. Let C be a cell in form-based program P, and let F be the formula for C. We treat each simple expression in F that sets the value of C as a *definition* of C. We treat each simple expression in F that refers to another cell D as a *use* of D. As with imperative languages [24], we define two types of uses. A *predicate use* (p-use) occurs in a predicate expression, and directly affects predicate outcome. A *computation use* (c-use) occurs in a nonpredicate expression, and affects the computation or display but does not directly affect predicate outcome. Let $\bar{F}$ be the formula graph for F. We associate each c-use with the node in $\bar{F}$ that corresponds to its expression, and we associate each p-use with each of the two edges in $\bar{F}$ that connect its associated predicate node to that node's successors. For simplicity, we refer to these nodes or edges themselves as "definitions," "c-uses," and "p-uses."

To test interactions between cells in form-based programs, we trace the flow of data between definitions and uses of those cells. Our goal is to determine *definition-use associations*, which pair definitions of cells with uses that those definitions may reach. Adapting the approach of [11], we consider two types of definition-use associations. A *definition-c-use association* is a triple (n_1, n_2, C), where n_1 is a definition of cell C (a node in C's formula graph), n_2 is a c-use of cell C (a node not in C's formula graph), and there is a valid cell evaluation order under which the definition of C in n_1 can reach the c-use of C in n_2. A *definition-p-use association* is a triple (n_1,(n_2, n_3),C), where n_1 is a definition of cell C (a node in C's formula graph), (n_2, n_3) is a p-use of cell C (a node not in C's formula graph), and there is a valid cell evaluation order under which the definition of C in n_1 can reach the p-use of C in n_2, and cause the associated predicate to be evaluated such that n_3 is the next node reached. Note that these definitions specify only *executable* definition-use associations.

With imperative programs, we can conservatively identify executable definition-use associations by using dataflow analysis algorithms that propagate definitions along control flow paths (e.g., [1, 12, 22, 26]). This approach does not directly apply to form-based programs, because in form-based programs the "flow of control" associated with evaluation orders of cells is not explicit, and varies over different evaluation engines. However, in non-recursive form-based programs such as those we are considering, the syntax of cell formulas and the restriction that cell C may only be defined in C support a simpler approach. In such programs, every definition of C in the formula for C reaches (statically) every use of C in the program. An efficient "scan-and-link" algorithm conservatively identifies executable definition-use associations in P by scanning formulas, locating definitions and uses in those formulas, and pairing each definition of each cell with each use of the cell.

To utilize definition-use associations for test adequacy purposes, we must define what it means to exercise an association. Given definition-use association (n_1, n_2,C) (or (n_1,(n_2, n_3),C)), where n_2 (or (n_2, n_3)) is contained in formula graph $\bar{F}$ corresponding to formula F, we say that a test t *exercises* that association if t causes evaluation of F, and that evaluation traverses a path through $\bar{F}$ that contains n_2 (or (n_2, n_3)). This definition of "exercises" differs from the analogous definition for imperative programs. There, a definition-use association (n_1, n_2, d) is exercised by a test t only if t executes a path that first reaches n_1, and subsequently reaches n_2 without encountering a redefinition of d. Here, because a test commences with a current set of definitions "in force," the effect of these definitions on uses can be evaluated without reexecuting the expression that contains the definition. To execute definition-use association (n_1, n_2, d) it is necessary only to execute n_2, under the condition in which the current definition of d has been provided by n_1; that definition is "pulled across" and participates in the computation. This definition of exercises facilitates incremental testing, and also, lets us define dataflow adequacy criteria independently of evaluation engines.

For imperative programs, a wide range of dataflow test adequacy criteria have been defined; each yields a particular coverage of definition-use associations. For example, we can require that all/some definitions reach all/some of their associated uses/c-uses/p-uses, via all/some of the paths over which the definitions can possibly reach those uses [11]. Or, we can require that chains of associations of various lengths be executed [21]. Many, but not all, of these criteria are directly applicable to form-based programs. We focus on the "all uses" dataflow test adequacy criterion.

For imperative programs, the *all uses* criterion requires test data to exercise each executable definition-use association that occurs in the program by at least one path. For form-based programs, where our definition of what it means to exercise a definition-use association does not involve paths from definitions to uses, the criterion simply requires test data to "exercise each executable association."

Table 4 depicts a test suite that is all-uses adequate for `rootsolver`. The first four tests in this suite are node- and edge-adequate; to achieve all-uses adequacy, test 5 was added. As the example illustrates, the all-uses criterion forces us to execute tests that exercise interactions between cells that were not exercised by node-adequate or cell-dependence-adequate test suites. By doing so, the criterion can help uncover errors that may remain undetected by node, edge, or cell-dependence criteria. For example, the criterion requires us to exercise definition-use associations ($e5$,$f7$,e) and ($e5$,$g7$,e); in doing so, we detect the error in $e5$ that was described in Section 4.3.

The superiority of all-uses adequacy illustrated in this example reflects a relationship that also appears to exist in the imperative-language counterparts of these criteria; a relationship that has been examined both analytically and empirically (e.g., [11, 15]). Where the application of all-uses adequacy to form-based programs is concerned, however, an additional advantage involves its (relative) ease of application. Dataflow analysis and dataflow testing are complicated for imperative programs by the presence of dynamically determined addressing in forms such as dynamic array indexing or pointer accesses. These complications force dataflow analysis and testing techniques to incur imprecision for imperative programs. Form-based programs may utilize arrays and matrices and refer to them in formulas; however, for most form-based languages, such references can be resolved statically. Form-based programs may have aliases, to the extent that multiple names may refer to a single cell; however, for most form-based languages these aliases, too, can be resolved statically. Thus, for programs in most form-based languages, definition-use associations can be more precisely determined than for programs in imperative languages, resulting in more precise test data.

Another potential advantage involves nonexecutable definition-use associations. There is no algorithm to determine whether an arbitrary definition-use association can be executed; determining executability can account for a substantial amount of the effort involved in using dataflow criteria in practice [30]. Our preliminary experience with

Test	Inputs	Def-use coverage
1	{a=1, b=-3, c=2}	(a2,d2,a),(b2,d2,b),(c2,d2,c),(d2,(e2,e3),d), (d2,e3,d) (a2,(f2,f4),a),(a2, f7,a),(d2,(f4,f7),d),(b2,f7,b),(e3, f7,e) (a2,(g2,g4),a),(a2, g7,a),(d2,(g4,g7),d),(b2,g7,b),(e3, g7,e)
2	{a=0, b=3, c=3}	(a2,d2,a),(b2,d2,b),(c2,d2,c),(d2,(e2,e3),d), (d2,e3,d),(a2,(f2,f3),a) (a2,(g2,g3),a),(b2,(f3,f5),b),(b2,(g3,g5),b),(b2,f5,b), (b2,g5,b),(c2,f5,c), (c2,g5,c)
3	{a=0, b=0, c=3}	(a2,d2,a),(b2,d2,b),(c2,d2,c),(d2,(e2,e4),d) (d2,(e4,e5),d),(a2,(f2,f3),a), (a2,(g2,g3),a), (b2,(f3,f6),b), (b2,(g3,g6),b)
4	{a=1, b=1, c=2}	(a2,d2,a),(b2,d2,b),(c2,d2,c),(d2,(e2,e4),d), (d2,(e4,e6),d),(a2,(f2,f4),a), (a2,(g2,g4),a),(d2,(f4,f8),d), (d2,(g4,g8),d)
5	{a=1, b=2,c=1}	(a2,d2,a),(b2,d2,b),(c2,d2,c),(d2,(e2,e4),d), (d2,(e4,e5),d), (a2,(f2,f4),a),(d2,(f4,f7),d),(a2,(g2,g4),a),(d2,(g4,g7),d) (a2,f7,a),(b2,f7,b),(e5,f7,e),(a2,g7,a),(b2,g7,b),(e5,g7,e)

Table 4. Definition-use adequate test suite for `rootsolver`**.**

form-based programs, however, suggests that they contain few nonexecutable definition-use associations. The absence of loops and of definitions that "kill" other definitions may partially account for this. Moreover, we believe that in practice, nonexecutable definition-use associations in form-based programs indicate places where users have made errors in formulas. If this is true, determining nonexecutable definitions may directly aid validation. We will investigate this conjecture further in empirical studies.

5. Conclusions and Future Work

We have described issues in, and strategies for, the testing of form-based visual programs, and presented test adequacy criteria for use in application to those programs. We have shown that an analogue to the traditional "all-uses" data-flow testing criteria is well suited for code-based testing of form-based visual programs: it provides important error-detection ability, and can be applied more easily to these programs than to imperative programs.

To empirically investigate the testing strategies described in this paper, we are implementing a testing environment called WYSIWYT ("What You See Is What You Test"), in the context of the Forms/3 visual programming system. This environment will let a Forms/3 programmer enter a "testing mode", in which the engine calculates definition-use associations in an active Forms/3 program, and records definition-use associations as they are executed. When a programmer adds, deletes, or modifies a formula, the engine will incrementally update its record of definition-use associations, determine associations that may be affected by the change, and record them as now requiring revalidation. In this way, the environment will support the application of an integrated, incremental program development and validation environment for Forms/3 programs. In keeping with the form-based visual language paradigm, the environment will present testing-related data visually; users will be able to view definition-use associations that have been satisfied thus far, satisfied on the previous input, or remain to be satisfied, by clicking on cells. We will use this environment to empirically study the fault-detection abilities and usability of our strategies.

References

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

[2] A. Ambler, M. Burnett, and B. Zimmerman. Operational versus definitional: A perspective of programming paradigms. *Computer*, 25(9), Sept. 1992.

[3] J. Atwood, M. Burnett, R. Walpole, E. Wilcox, and S. Yang. Steering programs via time travel. In *1996 IEEE Symposium on Visual Languages*, pages 4–11, Sept. 1996.

[4] A. Azem, F. Belli, O. Jack, and P. Jedrzejowicz. Testing and reliability of logic programs. In *The Fourth International Symposium on Software Reliability Engineering*, pages 318–327, 1993.

[5] F. Belli and O. Jack. A test coverage notion for logic programming. In *The 6th International Symposium on Software Reliability Engineering*, pages 133–142, 1995.

[6] P. Brown and J. Gould. Experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems*, 5(3):258–272, July 1987.

[7] M. Burnett and A. Ambler. Interactive visual data abstraction in a declarative visual programming language. *Journal of Visual Languages and Computing*, 5(1), Mar. 1994.

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

[9] R. Creeth. Micro-computer spreadsheets: Their uses and abuses. In *Journal of Accounting*, pages 90–93, June 1985.

[10] F. Del Frate, P. Garg, A. Mathur, and A. Pasquini. On the correlation between code coverage and software reliability. In *The Sixth International Symposium on Software Reliability Engineering*, pages 124–32, Oct. 1995.

[11] P. G. Frankl and E. J. Weyuker. An applicable family of data flow criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, Oct. 1988.

[12] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, Mar. 1994.

[13] J. Horgan, S. London, and M. Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–9, Sept. 1994.

[14] C. E. Hughes and J. M. Moshell. *Visual Languages and Applications*. Plenum Publishing, New York, 1990.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *The 16th International Conference on Software Engineering*, pages 191–200, May 1994.

[16] A. Kay. Computer Software. *Scientific American*, pages 53–59, Sept. 1984.

[17] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, May 1993.

[18] C. Lewis. NoPumpG: Creating interactive graphics with spreadsheet machinery. In E. Glinert, editor, *Visual Programming Environments: Paradigms and Systems*, pages 526–546. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[19] G. Luo, G. Bochmann, B. Sarikaya, and M. Boyer. Control-flow based testing of prolog programs. In *The 3rd International Symposium on Software Reliability Engineering*, pages 104–113, 1992.

[20] B. A. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of CHI '91: Human Factors in Computing Systems*, pages 243–249, Apr. 1991.

[21] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, 10(6), Nov. 1984.

[22] H. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations in C programs. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[23] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, fourth edition, 1997.

[24] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, Apr. 1985.

[25] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr. 1997.

[26] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, Jan. 1988.

[27] T. J. Smedley, P. T. Cox, and S. L. Byrne. Expanding the Utility of Spreadsheets Through the Integration of Visual Programming and User Interface Objects. In *Advanced Visual Interfaces '96*, May 1996.

[28] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 15(4):465–470, 1982.

[29] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12(12):1128–1138, Dec. 1986.

[30] E. J. Weyuker. More experience with dataflow testing. *IEEE Transactions on Software Engineering*, 19(9), Sept. 1993.

[31] N. Wilde and C. Lewis. Spreadsheet-based interactive graphics: from prototype to tool. In *Proceedings of CHI '90: Human Factors in Computing Systems*, pages 153–159, Apr. 1990.

[32] N. P. Wilde. A WYSIWYC (What You See Is What You Compute) spreadsheet. In *1993 IEEE Symposium on Visual Languages*, pages 72–76, Aug. 1993.

[33] W. Wong, J. Horgan, S. London, and A. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *The Fifth International Symposium on Software Reliability Engineering*, pages 230–8, Nov. 1994.

[34] M. M. Zloof. Query by example: a database language. *IBM Systems Journal*, 16(4), 1977.