Proceedings of the 20th Int'l Conference on Software Engineering, Kyoto, Japan, April, 1998, pages 198-207

What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs

Gregg Rothermel, Lixin Li, Christopher DuPuis, Margaret Burnett
Department of Computer Science
Oregon State University
Corvallis, OR 97331
{grother,lili,dupuis,burnett}@cs.orst.edu

ABSTRACT
Form-based visual programming languages, which include commercial spreadsheets and various research systems, have had a substantial impact on end-user computing. Research shows, however, that form-based visual programs often contain faults. We would like to provide at least some of the benefits of formal testing methodologies to the creators of these programs. This paper presents a testing methodology for form-based visual programs. To accommodate the evaluation model used with these programs, and the interactive process by which they are created, our methodology is validation-driven and incremental. To accommodate the users of these languages, we provide an interface to the methodology that does not require an understanding of testing theory. We discuss our implementation of this methodology and empirical results achieved in its use.

KEYWORDS
software testing, visual programming, spreadsheets

1 INTRODUCTION

Form-based visual programming languages provide a declarative approach to programming, characterized by a dependence-driven, direct-manipulation working model [1]. Users of form-based languages create cells, and define formulas for those cells. These formulas reference values contained 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.

Form-based visual programming languages include, as a subclass, commercial spreadsheet systems. These systems are widely used by end-users, for a variety of computational tasks. The form-based visual language paradigm is also a subject of ongoing research. For example, there is research into using form-based languages for matrix manipulation problems [25], for providing steerable simulation environments for scientists [4], and for specifying full-featured GUIs [16].

Despite the end-user appeal of form-based languages and the perceived simplicity of the paradigm, research shows that form-based visual programs often contain faults. For example, in one empirical study of experienced spreadsheet users [2], 44 percent of the spreadsheets created by those users were found to contain user-generated faults. Compounding this problem, creators of spreadsheets express unwarranted confidence in the reliability of their programs [28]. In spite of this evidence, we find no discussion in the research literature of techniques for testing form-based visual programs.

In previous work [23], we discussed strategies for testing form-based visual programs. We showed that significant differences exist between the form-based and imperative programming paradigms, and that these differences have ramifications for testing methodologies. These differences can be divided into three classes.

The first class of differences pertains to evaluation models. Evaluation of form-based programs is driven by data dependencies that exist between cells, and form-based programs contain explicit control flow only within cell formulas. Thus, form-based programs are more appropriately tested using adequacy criteria that are data-dependence-based than criteria that are strictly control-flow-based. The dependence-driven evaluation model also implies that evaluation engines have considerable flexibility in the scheduling algorithms and optimization devices they might employ to perform computations. A methodology for testing form-based programs must be compatible with such mechanisms, and not rely upon particular evaluation orders or prevent optimizations based on value caching.

The second class of differences pertains to interactivity: form-based programming environments 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 of spreadsheets. 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 programs while maintaining suitable response time.

The third class of differences pertains to the users of form-based languages. Imperative languages are most commonly used by professional programmers who are in the business of producing software. These programmers 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, form-based programming environments are used by a variety of users, many of whom are not professional programmers and have no interest in learning about formal testing methodologies. Our goal is to provide at least some of the benefits of formal testing methodologies to these users.

Building on strategies discussed in [23], this paper presents a testing methodology for form-based visual programs. To accommodate the evaluation models used with these programs, and the interactive process by which they are created, our methodology is validation-driven and incremental. This is accomplished through a test adequacy criterion that focuses on dependencies that influence validated output cells, and by the use of incremental program analysis. 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 form-based language environment to provide testing information visually.

2 BACKGROUND

Form-based visual languages.
Users of form-based languages set up forms and specify their contents in order to program. The contents of a form are a collection of cells; each cell's value is defined by that cell's formula, and as soon as the user enters a formula, it is evaluated and the result is displayed. The best-known examples of form-based languages are commercial spreadsheets, but there are also many research systems (e.g. [3, 16, 24, 25]) based on this paradigm.

In this paper, we present examples of form-based programs in the research language Forms/3 [3, 10]. Figure 1 uses a traditional spreadsheet style to calculate student grades. Figure 2 shows how a user would construct a graphical clock in Forms/3. View (a) 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

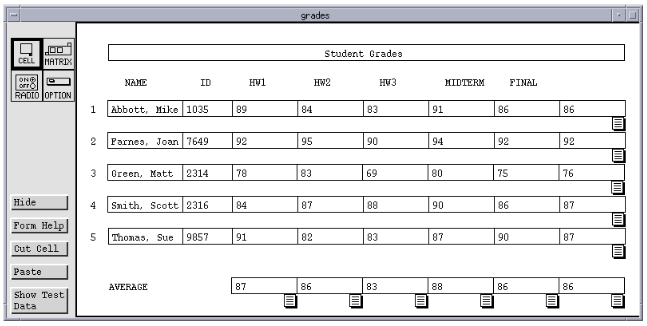


Figure 1. *Spreadsheet for calculating student grades.*

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 formula tabs, borders, and cells that calculate intermediate results can be hidden, and cells rearranged, to reach the user view shown in (b).

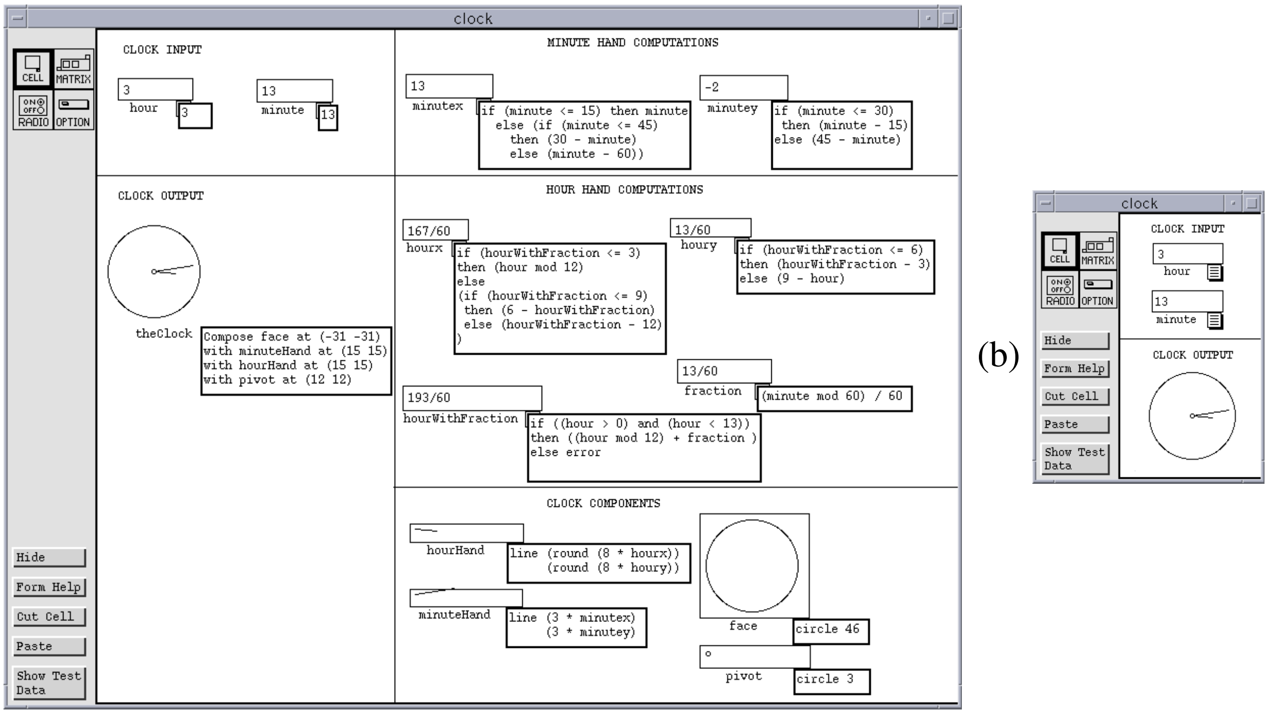
In this paper, we consider a subset of Forms/3 that is representative of "pure" form-based visual languages: those with no macros or imperative sublanguages and no recursion. The subset includes ordinary spreadsheet-like formulas for mathematics and conditional operations, and support for elementary graphics. The grammar for the formulas in this subset is shown in Table 1. The Figures were programmed using this subset. 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.

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 ::= + | − | ∗ | / | AND | OR | = | ...
unaryPrefixOperator ::= NOT | ERROR? | CIRCLE | ...
binaryPrefixOperator ::= LINE | BOX | ...
withclause ::= WITH subExpr AT (subExpr subExpr) | WITH subExpr AT (subExpr subExpr) withclause

Table 1. *Grammar for formulas*

An abstract model for form-based programs.
Test adequacy criteria provide help in selecting test data and in deciding whether a program has been tested "enough." Test adequacy criteria have been well researched for imperative languages (e.g. [6, 9, 17, 19]), where they are often defined on abstract models of programs rather than on code itself. In [23] we presented such a model for form-based languages; we call our model a *cell relation graph* (CRG). Figure 3 depicts a partial CRG for `Clock`.

(a)



(b)

Figure 2. *Programming a clock in Forms/3.*

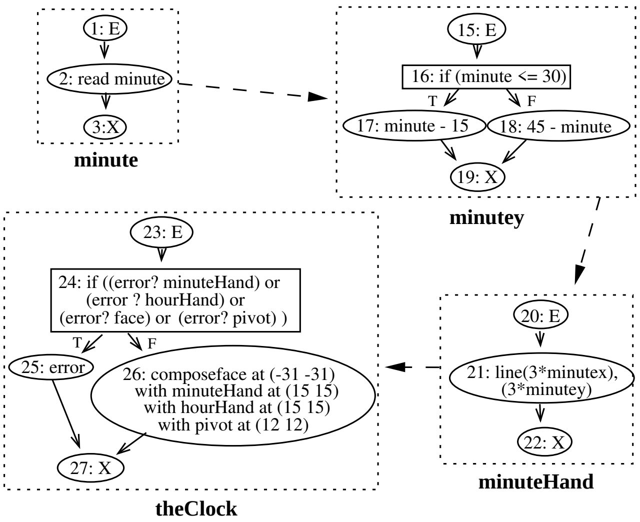


Figure 3. *Partial cell relation graph for* `Clock`.

A CRG uses two sets of components to model two aspects of form-based programs. The first set of components, *formula graphs*, model flow of control within cell formulas, and are comparable to the control flow graphs used to represent procedures in imperative programs. Figure 3 shows the formula graphs for four cells, delimited by dotted rectangles. In the figure, nodes labeled "E" and "X" are *entry* and *exit* nodes, respectively, and represent initiation and termination of the evaluation of formulas. Nodes with multiple out-edges (represented as rectangles) are *predicate* nodes. Other nodes are *computation* nodes. Edges represent flow of control between expressions; edge labels indicate the value to which conditional expressions must evaluate for that control path to be taken.

The second set of components in the CRG, *cell dependence edges*, model dependencies between cells. Figure 3 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. For readability, the figure depicts these edges as beginning and terminating at the rectangles that delimit formula graphs; in fact these edges begin and terminate at the entry and exit nodes of those graphs.

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.

We used this abstract model to define several test adequacy criteria for form-based programs. We showed that a criterion based on the *all-uses* dataflow adequacy criterion defined originally for imperative programs (e.g., [14, 17, 21]), which relates test adequacy to interactions between occurrences of variables in the source code, is particularly appropriate for form-based programs because it exercises both interactions between cells and expressions within cell formulas. In this paper, we restrict our attention to this criterion.

DU-adequacy for form-based programs.
In form-based programs, 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 program P, with formula F and formula graph $\bar{F}$. 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.

A *definition-use association* (du-association) links definitions of cells with uses that those definitions can reach. Two types are of interest. A *definition-c-use association* is a triple (n_1, n_2, C), where n_1 is a definition of cell C, n_2 is a c-use of C, and there exists an assignment of values to P's input cells, in which n_1 reaches 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, (n_2,n_3) is a p-use of C, and there exists an assignment of values to P's input cells, in which n_1 reaches n_2, and causes the predicate associated with n_2 to be evaluated such that n_3 is the next node reached.

To preserve the applicability[1] of adequacy criteria based on these definitions, the definitions specify only *executable* du-associations: du-associations for which there exists some input that causes the definition to reach the use. Determining whether a du-association is executable is, however, a difficult problem [9, 27]; thus, algorithms for calculating the du-associations that exist in a program typically conservatively approximate them, by collecting the du-associations that appear (statically) to exist in the code. We discuss this issue further in [23] and Section 4.

Following the notion of an "output-influencing all-du-pairs" criterion introduced by [7], we define a test adequacy criterion in terms of du-associations that affect cell outputs. A test suite T is *du-adequate* for program P if and only if, for each du-association *dua* in P, there exists at least one test t in T that exercises *dua* in such a way that *dua* influences, directly or indirectly, a cell output.

3 A METHODOLOGY FOR TESTING FORM-BASED VISUAL PROGRAMS

In Section 1, we described three classes of differences between the form-based visual language paradigm and traditional imperative paradigms. To accommodate these differences, we have developed a testing methodology that is validation-driven and incremental, and integrated at a fine granularity into the programming environment, providing the following functionalities:

- The ability to incrementally determine the static du-associations in an evolving program whenever a new cell formula is entered.
- The ability to automatically track execution traces, which provide the information necessary to determine the dynamic du-associations that currently influence calculations.
- A user-accessible facility for pronouncing outputs "validated" at any point during program 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 program is.
- The ability to determine the du-associations affected by a program change, and immediately depict their altered validation status in the visible section of the program.
- The ability to recalculate du-associations and validation information when an entire pre-existing program is loaded, or when a large portion of a program is modified by a single user action.

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

Task 1: Collecting static du-associations.
Suppose that, starting with an empty form, the user begins to build the `Clock` application discussed in Section 2 by entering cells and formulas, reaching the state shown in Figure 4. Assume for the moment that the user does not change any formulas, but simply continues to add new ones. (We remove this restriction later.)

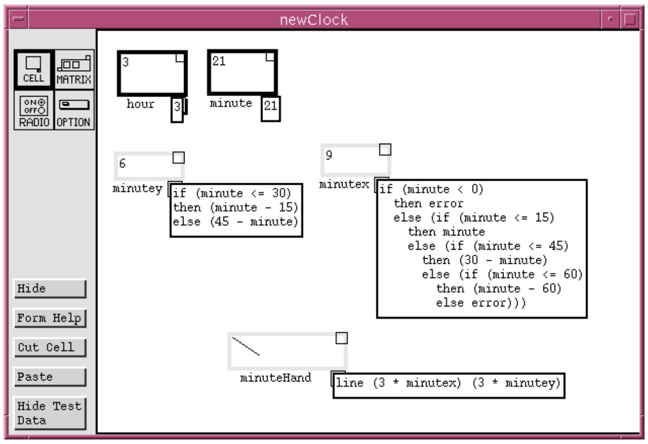


Figure 4. `Clock` *at an early stage. Part of the program has been entered.*

Because it would be expensive to exhaustively compute the du-associations for the entire program after each new formula is added, we compute them incremen-

[1] 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 [26].

tally. Several algorithms for incremental computation of data dependencies exist for imperative programs (e.g., [15, 20]), and we could adapt one of these algorithms to our purpose. However, there are two attributes of form-based programming environments that allow a more efficient approach.

First, in non-recursive form-based languages, the syntax of cell formulas and the fact that C can only be defined in its own formula ensure that every definition of C reaches (statically) every use of C in the program. Second, in form-based programming environments, 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 and all cells that are directly referenced by the new cell.[2] At this time, the engine can 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 facts mean that we can incrementally collect du-associations following the addition of a cell C by associating all definitions in C with all uses of C in cells that reference C, and associating all definitions in cells that C references with all uses of those cells in C.[3]

Our prototype uses a hash table to efficiently store the following data for each cell C: $C.CellsThatRef$, the cells that reference C; $C.CellsRefedBy$, 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 use is described later; $C.DUA$, a set of pairs (*du-association*, *exercised*) for each static du-association (d, u) such that u is in $C.LocalUses$, and *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 for 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 program edit. The remaining items are calculated by the testing subsystem.

Algorithm `CollectAssoc` of Figure 5 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 referring cells' uses of C.

For example, referring back to Figure 4, suppose that the most recent formula entered is that for cell `minuteY`.

1. **algorithm** `CollectAssoc`(C)
2. **for** each cell $D \in C.CellsRefedBy$ **do**
3.   **for** each definition d (of D) $\in D.LocalDefs$ **do**
4.   **for** each use u of $D \in C.LocalUses$ **do**
5. $C.DUA = C.DUA \cup \{((d,u),\texttt{false})\}$
6. **for** each cell $D \in C.CellsThatRef$ **do**
7.   **for** each use u of $C \in D.LocalUses$ **do**
8.   **for** each def d (of C) $\in C.LocalDefs$ **do**
9. $D.DUA = D.DUA \cup \{((d,u),\texttt{false})\}$

Figure 5. *Algorithm for collecting du-associations.*

Note that its value is displayed, even though the program has not been completely entered; when the evaluation engine was triggered to display this value, it collected $C.CellsThatRef$, $C.CellsRefedBy$, $C.LocalDefs$, and $C.LocalUses$ for `minuteY` (as it had previously done 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 3 as: (2,(16,17),`minute`), (2,(16,18),`minute`), (2,17,`minute`), (2,18,`minute`), (17,21,`minutey`), and (18,21,`minutey`).

`CollectAssoc` runs in time $O(udn)$, where n is the number of cells that directly reference or are referenced by C, and u and d are the maximum number of uses and definitions, respectively, in those cells. In practice, u and d are typically small, bounded by the number of references in a single formula – usually less than 10. In this case the algorithm's time complexity is of the same order as the evaluation engine's cell traversal needed to maintain a correct display and process cached values when a new formula is added – the event that triggers `CollectAssoc`.

Task 2: Tracking execution traces.
To track execution traces, which enable the incremental computation of du-associations that have been exercised, we have simply inserted a probe into the evaluation engine. When cell C executes, this probe records the execution trace on C's formula graph, storing it in $C.Trace$. For example, in the case of `Clock`, at the moment depicted in Figure 4, the execution trace stored for cell `minutey`, described in terms of Figure 3's node numbers, is (15,16,17,19). If the cell is subsequently reevaluated, the system replaces the old execution trace with the new one. This approach functions for all varieties of evaluation engines: whether the engine eagerly or lazily evaluates cells, following any input and any dependence-preserving evaluation sequence, all cells have associated with them their most recent execution trace.

Storing only the most recent execution trace in $C.trace$ is sufficient because the cumulative coverage in $C.DUA$ is updated incrementally during validation, as described in our discussion of Task 3.

[2]This is true for both eager and lazy form-based languages, because even if a cell's recalculation can be deferred, it could have a cached value that must be marked "dirty" to indicate that it is now invalid. Value caching is necessary for efficient display maintenance; most form-based languages use it to some extent.

[3]See [15] for a different view of incremental computation of du-associations as applied within the imperative language paradigm.

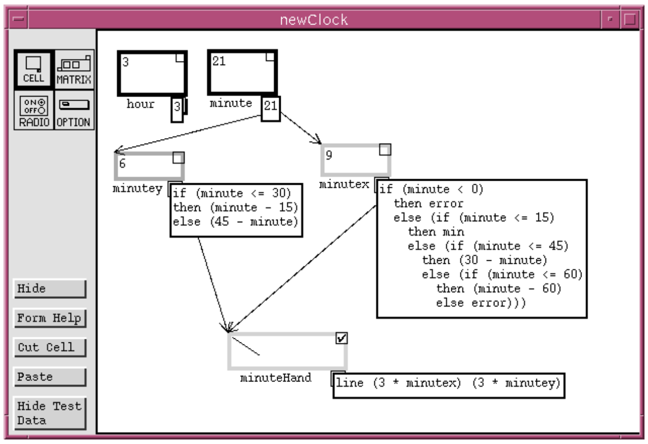


Figure 6. *The evolving* `Clock` *program at an early stage, after the* `minuteHand` *cell has been validated.*

Task 3: Pronouncing outputs "validated".
In this section, we show how we use the data collected in Tasks 1 and 2 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 program.

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 6 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 (not shown in the figure), which means the current value has not been validated but some previously-displayed value has. (The evaluation engine makes this change while visiting cells affected by the new input.) The third possible appearance, a blank validation tab, means no validations have been done since the last formula change to C or to a non-input 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.

A finer-grained device for communicating testing status involves test adequacy. Whenever a du-association participates in the production of a validated value, we set 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) to "`true`". We then calculate the percentage of the du-associations, whose uses occur in the cell, that have been exercised. We use this percentage 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 paper, 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 6, the computation of `minuteHand`'s value involves 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, after the user validates `minuteHand`, the cell borders are darkened using these fractions. Input cells (those with constant formulas) 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., executed with validation) at least once. As the figure shows, the user can also display arrows that show all the cell reference patterns (du-associations) at the granularity of cells; we are currently implementing these arrows at the subexpression level with the same color scheme as the borders, to explicitly identify which cell reference patterns still need to be tested.

Figure 7 displays our algorithm `Validate`, which is invoked when the user pronounces a displayed value valid. The algorithm uses the static 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 participate in the production of C's current value, and to update borders of participating cells.[4] As the algorithm proceeds, it adds to stored $.DUA$ data that indicates the du-associations that have been validated 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.

In this algorithm, the use of *ValidatedID* ensures that the algorithm terminates in worst-case time proportional to the number of du-associations validated, rather than to the size of the program. This is the same order as the cost of calculating the cell's value, but the algorithm is not triggered at that time, so, unlike the other algorithms we have presented, its cost is not masked by the cost of the evaluation process. *ValidatedID* is set to 0 when the programming environment is first activated.

[4] A generalization of this algorithm related to the approach of [7] 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 works for programs with recursion, iteration, and redefinitions of variables. For most form-based languages, however, the more efficient `Validate` approach suffices.

```
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. UpdateBorder(C)
```

Figure 7. *Algorithm for updating test adequacy information following a validation request.*

When cells are created or added to the system, their *.ValidatedID* fields are initialized to 0. On each invocation of `Validate`, *ValidatedID* is incremented (line 1). The *.ValidatedID* fields for all cells visited are assigned this value of *ValidatedID*, which prevents duplicate visits to the same cell.[5]

Task 4: Adjusting test adequacy information. So far, we have focused on how the system handles cell formulas as they are added to a program. We now consider the other basic edits possible with form-based programs, 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 (that may or may not be followed by validation requests), 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 non-constant 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 program is still fairly well tested, and not realize the extent to which the modification invalidates previous testing.

To address this lack of awareness, the system must immediately reflect the new test adequacy status of the program whenever a cell is modified.[6] To accomplish this, the system must (1) update C's static du-association and dynamic 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 colors to reflect the new "testedness" of affected cells. We must also adjust validation tab statuses on visited cells, changing all checkmarks and questionmarks to questionmarks if the cell retains any exercised du-associations after affected associations have been reset, or to blank if all the cell's exercised flags are now unset. For example, in the completed `Clock` program, 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 reinitialized.

```
algorithm UnValidate(C)
AffCells = {}
UnValidatedID = UnValidatedID +1
UnValidateCell(C)
for each cell D ∈ AffCells do
    UpdateBorder(D)
    UpdateValTab(D)

procedure UnValidateCell(C)
C.UnValidatedID = UnValidatedID
AffCells = AffCells ∪ C
for each cell D ∈ C.CellsThatRef do
  for each definition d (of C) in C do
    for each ((d,u),true) ∈ D.DUA do
      D.DUA = D.DUA ∪{((d, u),false)}−{((d, u),true)}
  if D.UnValidatedID < UnValidatedID then
    UnValidateCell(D)
```

Figure 8. *Algorithm for updating test adequacy information following a modification.*

Our subsystem 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 affected cells. The algorithm, `UnValidate`, given in Figure 8, is similar to `Validate`, but instead of using dynamic information to walk backward through the program, it uses static information to walk forward. As the algorithm walks forward, it changes the *exercised* flag on each previously exercised 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 color and validation tab for each cell in *AffCell*.

At this point, the static du-association and dynamic trace information stored with C can be updated. First, all stored static du-associations involving C are deleted; the environment can find these easily in the information stored for C and for cells in C.*CellsThatRef* and just delete them; this removal also guarantees that du-associations that end in C are no longer marked "exercised." Having removed the old du-associations, we need only re-invoke `CollectAssoc` as described in

[5] 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 program on each invocation. We assume that *ValidatedID* will not overflow, to simplify the presentation.

[6] In this context, the problem of interactive, incremental testing of form-based programs resembles the problem of regression testing imperative programs, and we could adapt techniques for incremental dataflow analysis (e.g., [15, 20]) and incremental dataflow testing (e.g. [11, 12, 22]) of imperative programs to generalize this approach. This generalized approach applies to programs in which cell references are recursive or in which formulas contain iteration. For most form-based languages, however, the simpler approach that we present here suffices.

Section 3 to add new associations. Finally, stored execution traces are automatically updated via the evaluation engine as described earlier.

As was the case with Tasks 1 and 2, the cell visits required by `UnValidate` are already required for display and value cache maintenance; therefore 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.

Task 5: Batch computation of information.
There are some circumstances in which it may be necessary to calculate static definition-use information for a whole program or section of a program – for example, if the user does a block copy/paste of cells, or imports a program from another environment that does not accumulate necessary data. One possible response to such an action is to iteratively call the algorithms presented so far — which are written for single cell changes — for each new, modified or deleted cell in the new program section. Although we do not present it here due to space limitations, we use a more efficient algorithm that takes an entire set of cells as input, and makes passes over this set to update information on du-associations and validation status.

4 EMPIRICAL RESULTS

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 form-based programs. To empirically address this issue, we have implemented a prototype within the Forms/3 programming environment. Our prototype incorporates all of the algorithms described in this paper except for that of Figure 8, which is partially complete. The screen shots used in this paper are from this prototype. We have used our prototype to perform an empirical study of the effectiveness of du-adequate test suites at detecting faults.

Methodology.
For our study, we obtained eight Forms/3 programs (see Table 2) from experienced Forms/3 users. Three of the programs (TimeCard, Grades, and Sales) are typical of spreadsheet programs, and the others are typical of form-based programs written in research languages: two are simple simulations (FitMachine and MicroGen), one a graphical desktop clock (Clock), one a number-to-digits splitter (Digits), and the last a quadratic equation solver (Solution).

We asked seven users experienced with Forms/3 and commercial spreadsheets to insert faults into our subject programs which, in their experience, are representative of faults found in Forms/3 programs or in spreadsheets.

Program	*Expr*	*DUA*	*Ver*	*Pool Size*	*Suite Size*
Clock	33	64	7	250	11.3
Digits	35	89	10	230	22.7
FitMachine	33	121	11	367	30.2
Grades	61	55	10	80	9.8
MicroGen	16	31	10	170	10.4
Sales	30	28	9	176	10.4
Solution	20	32	11	99	12.0
TimeCard	33	92	8	240	16.7
average	33	64	10	202	15.4

Table 2. *Data about experimental subjects, including (from left to right) program name, number of expressions in the program, number of du-associations in the program, number of faulty versions, size of test pool, and average size of du-adequate test suites for the program.*

We then asked a Forms/3 user who had no knowledge of these specific faults to generate a pool of tests for each of the base versions of the subject programs. For each base program, this user first created tests of program functionality. He then executed these tests on the base program to determine whether together they exercised all executable du-associations in the program, and generated additional tests to ensure that each executable du-association in the program was exercised by at least 5 tests in the test pool. He also verified that for all tests, validated cells in the base version produced correct values.[7]

We used these test pools to create du-adequate test suites for our programs. To do this, we first determined, for each test t in the test pool, the du-associations exercised by t. We then created test suites by randomly selecting a test, adding it to the test suite only if it added to the cumulative coverage achieved by tests in that suite thus far, and repeating this step until coverage was du-adequate. We generated between 10 and 15 du-adequate test suites for each of our subject programs; Table 2 lists the average sizes of these test suites.

Because the base version was known to produce correct output, and because only a single fault was inserted in each faulty version, we could determine whether a fault had been revealed in a modified version P' by a test suite T simply by comparing the validated output of P' (the output which, for that test, was confirmed by the tester to be correct) for each test t in T with the validated output of P on t. Thus, to obtain fault detection results, for each base version P, with its faulty versions $P_1 \ldots P_k$ and universe U of test suites, for each test suite T in U, we:

1. ran all tests in T on P, saving outputs,
2. for each modified version P_i of P:

[7] This procedure was modeled after a recent study of imperative program testing [13].

(a) ran all tests in T on P_i, saving outputs,

(b) recorded T as fault-revealing for P_i if and only if the output of the validated cell for some test t in T executed on P_i differed from the output of that cell when t was executed on P.

Data and analysis.
Figure 9 displays fault detection data, using box plots to show, for each program, the percentage of faults detected by the du-adequate test suites. Dashed crossbars represent median percentages of faults detected over the set of test suites for the program. The boxes show the ranges of percentages in which half of the fault detection results occurred. The whiskers that extend below and above boxes indicate ranges over which the lower 25% and upper 25% of the data, respectively, occurred.

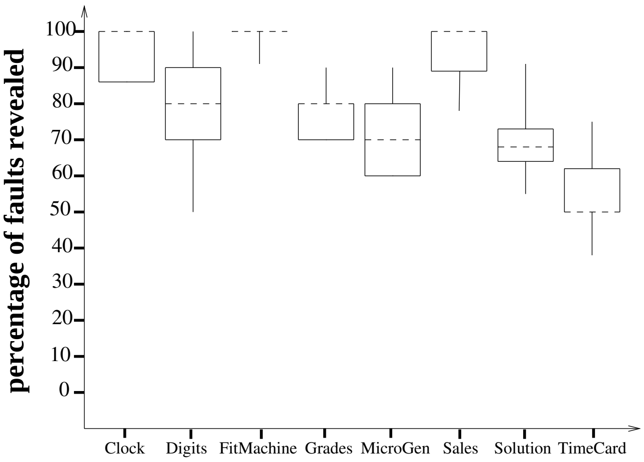


Figure 9. *Faults detected by the du-adequate test suites.*

The overall average (mean) percentage of faults detected for all programs, faulty versions, and test suites in our study was 81%. Fault detection varied across programs, but in all but one case (on two versions of TimeCard) exceeded 50%. Although differences in experimental instrumentation make comparisons difficult, this fault-detection effectiveness is comparable to or better than the effectiveness demonstrated by the all-uses criterion in studies of imperative programs [8, 13, 18, 27, 29].

These results are encouraging, but a few caveats are in order. Our subject programs are not large, and we have no data to show that they are representative of a larger class of form-based visual programs. Also, although our faulty versions involve manually-seeded faults created by experienced users, we cannot substantiate that these faults represent faults that occur in practice. Finally, we do not claim that our test suites are representative of those that would be constructed by typical users of form-based languages.

One cost factor associated with dataflow testing involves nonexecutable du-associations, which no inputs can exercise, but which are recognized by the testing system's static analysis. We discovered that on average, 11.65% of the du-associations calculated by our algorithms for our programs were nonexecutable. This rate is lower than the average rates of 26% and 27% observed in two studies of imperative programs reported in [27]. Nevertheless, the presence of these associations could be difficult to explain to users. Future work will consider whether techniques for determining (approximately) path feasibility (e.g. [5]) can operate cost-effectively behind the scenes to address this problem.

5 CONCLUSION

Due to the popularity of commercial spreadsheets, form-based visual languages are being used to produce software that influences important decisions. Furthermore, the use of this paradigm is likely to continue to grow, due to recent advances from the research community that expand its capabilities. 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 form-based 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 logically structured such that their work can be performed incrementally, and hence can be tightly integrated with the highly interactive environments that characterize form-based visual 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 is appropriate for use by a wide range of programmers, including the many end users who use spreadsheets, because it requires no knowledge of testing theory. Instead, the algorithms track the "testedness" of the program incrementally, and use visual devices to call attention to insufficiently tested interactions.

Our empirical results suggest that in practice, 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 form-based users may be substantial.

ACKNOWLEDGEMENTS

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. This work has been supported by the National Science Foundation under ASC 93-08649, by an NSF Young Investigator Award, and by Faculty Early CAREER Award CCR-9703108 to Oregon State University.

REFERENCES

[1] A. Ambler, M. Burnett, and B. Zimmerman. Operational versus definitional: a perspective on programming paradigms. *Computer*, 25(9):28–43, Sept. 1992.

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

[3] M. Burnett and A. Ambler. Interactive visual data abstraction in a declarative visual programming language. *J. Vis. Lang. and Comp.*, 5(1), Mar. 1994.

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

[5] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3), Sept. 1976.

[6] L. Clarke, A. Podgurski, D. Richardson, and S. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Softw. Eng.*, SE-15(11):1318–1332, Nov. 1989.

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

[8] P. Frankl and S. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Softw. Eng.*, 19(8):774–787, Aug. 1993.

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

[10] H. Gottfried and M. Burnett. Graphical definitions: Making spreadsheets visual through direct manipulation and gestures. In *IEEE Symp. Vis. Lang.*, Sept. 1997.

[11] R. Gupta, M. J. Harrold, and M. L. Soffa. Program slicing-based regression testing techniques. *J. Softw. Testing, Verif., and Rel.*, 6(2):83–112, June 1996.

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

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *16th Intl. Conf. Softw. Eng.*, pages 191–200, May 1994.

[14] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. Softw. Eng.*, 9(3):347–354, May 1993.

[15] T. Marlowe and B. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *ACM POPL*, pages 184–196, Jan. 1990.

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

[17] S. C. Ntafos. On required element testing. *IEEE Trans. Softw. Eng.*, 10(6), Nov. 1984.

[18] J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Softw. Practice and Experience*, 26(2):165–176, Feb. 1996.

[19] D. Perry and G. Kaiser. Adequate testing and object-oriented programming. *J. Object-Oriented Prog.*, 2, Jan. 90.

[20] L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Softw. Eng.*, 15(12):1537–1549, Dec. 1989.

[21] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, Apr. 1985.

[22] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proc. 1994 Intl. Symp. Softw. Testing and Analysis*, pages 169–184, Aug. 1994.

[23] G. Rothermel, L. Li, and M. Burnett. Testing strategies for form-based visual programs. In *The Eighth Intl. Symp. Softw. Rel. Eng.*, pages 96–107, Nov. 1997.

[24] T. Smedley, P. Cox, and S. Byrne. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *Adv. Vis. Int. '96*, May 1996.

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

[26] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. Softw. Eng.*, 12(12):1128–1138, Dec. 1986.

[27] E. J. Weyuker. More experience with dataflow testing. *IEEE Trans. Softw. Eng.*, 19(9), Sept. 1993.

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

[29] W. Wong, R. Horgan, S. London, and A. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th Intl. Conf. Softw. Eng.*, pages 41–50, Apr. 1995.