## AN ABSTRACT OF THE THESIS OF

Lixin Li for the degree of Master of Science in Computer Science presented on November 6,1997.

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

Redacted for privacy

Abstract approved: _____

Gregg Rothermel

Visual programming languages employ visual representation to make programming easier and make programs more reliable and more accessible. Visual program testing becomes increasingly important as more and more visual programming languages and visual programming environments come into real use. In this work, we focus on one important class of visual programming languages: form-based visual programming languages. This class of languages includes electronic spreadsheets and a variety of research systems that have had a substantial impact on end-user computing.

Research shows that form-based visual programs often contain faults, but 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 lack will hinder the real use of visual programming languages.

Our work addresses the lack of testing methodologies for form-based visual programs. In this document, we first examine differences between the form-based and imperative programming paradigms, discuss effects these differences have on method-

ologies for testing form-based programs, and analyze challenges and opportunities for form-based program testing.

We then present several criteria for measuring test adequacy 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 testing form-based visual programs: it provides important error-detection ability, and can be applied more easily to form-based programs than to imperative programs.

Finally, we present a testing methodology that we have developed 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 user base 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, its time costs, the mapping from our approach to the user interface, and empirical results achieved in its use.

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

by

Lixin Li

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed November 6,1997
Commencement June 1998

Master of Science thesis of Lixin Li presented on November 6,1997

APPROVED:

Redacted for privacy

Major Professor, representing Computer Science

Redacted for privacy

Chair of the Department of Computer Science

Redacted for privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Lixin Li, Author

## ACKNOWLEDGMENTS

I would especially like to express my gratitude to my major professor, Dr. Gregg Rothermel, for all the time and energy he has spent to improve my skills. Dr. Rothermel takes his role as mentor very seriously, and it is my great pleasure to work with such a professional.

My great thanks also go to Dr. Margaret Burnett, as my minor professor, she is always willing to offer good advice and give help. I feel lucky to have an advisor like her.

Thanks to everybody in the visual programming research group, particularly to Rebecca Walpole, John Atwood and Anurag Agrawal for helping me to understand the implementation of Forms/3. Thanks to Christopher Dupuis for implementing a graphical user interface for our testing methodology. Also thanks to everybody who participated in creating faulty programs for our empirical study.

# TABLE OF CONTENTS


Page

**Chapter 1: Introduction** 1

**Chapter 2: Background** 4

2.1 Software testing . . . . . 4

2.2 Form-based visual programming languages . . . . . 8

2.2.1 The form-based programming language paradigm . . . . . 8

2.2.2 Forms/3: a form-based visual programming language . . . . . 9

2.2.3 Evaluation strategies for form-based languages . . . . . 12

2.3 Related work . . . . . 14

**Chapter 3: Issues and opportunities** 16

3.1 Types of faults found in form-based programs . . . . . 16

3.2 Specification-based versus code-based testing . . . . . 17

3.3 Incremental versus nonincremental testing . . . . . 17

3.4 Applicability of criteria . . . . . 19

3.5 Integrated development, testing and debugging . . . . . 21

3.6 The user interface . . . . . 21

**Chapter 4: A methodology for testing form-based visual programs** 23

4.1 Adequacy criteria for form-based programs . . . . . 23

4.1.1 An abstract model for form-based programs . . . . . 25

4.1.2 Node and edge adequacy criteria . . . . . 27

4.1.3 The cell dependence adequacy criterion . . . . . 29

4.1.4 The du-validation criterion . . . . . 35

4.2 An incremental and validation-driven methodology . . . . . 36

4.2.1 Task1: Collecting static du-associations. . . . . 38

4.2.2 Task 2: Tracking dynamic du-associations. . . . . 42

4.2.3 Task 3: Pronouncing outputs "validated". . . . . 43

TABLE OF CONTENTS (Continued)

Page

4.2.4 Task 4: Adjusting test adequacy information. . . . . 46
4.2.5 Task 5: Batch computation of information. . . . . 48

**Chapter 5: Implementation and empirical study** **50**

5.1 Implementation . . . . . 50
5.1.1 General environment . . . . . 50
5.1.2 Primary data structures . . . . . 51
5.1.3 Main modules . . . . . 51
5.2 Empirical study . . . . . 52
5.2.1 Objectives . . . . . 52
5.2.2 Experimental design . . . . . 52
5.2.3 Data and analysis . . . . . 56

**Chapter 6: Conclusion** **60**

**Bibliography** **62**

**Appendix: Main Source Code for Implementation** **66**

## LIST OF FIGURES


| Figure | | Page |
|---|---|---|
| 2.1 | Control flow graph of program avg. | 6 |
| 2.2 | Spreadsheet to calculate student grades. | 9 |
| 2.3 | Programming a clock in Forms/3. | 10 |
| 2.4 | User's view of program `clock`. | 11 |
| 4.1 | Forms/3 program to obtain the root(s) of quadratic or linear equations. | 24 |
| 4.2 | Formula graph for cell `f` of `rootsolver`. | 26 |
| 4.3 | Cell relation graph for `rootsolver`. | 27 |
| 4.4 | `Clock`'s cell relation graph. | 39 |
| 4.5 | `Clock` at an early stage; part of the program has been entered. | 40 |
| 4.6 | Algorithm for collecting du-associations. | 41 |
| 4.7 | The evolving `Clock` program at an early stage, after the `minuteHand` cell has been validated. | 44 |
| 4.8 | Algorithm for updating test adequacy information following a validation request. | 46 |
| 4.9 | Algorithm for updating test adequacy information following a modification. | 49 |
| 5.1 | Faults detected by the du-validation-adequate test suites. | 57 |

## LIST OF TABLES


| Table | | Page |
|---|---|---|
| 2.1 | Node-adequate test suite for `avg`. | 7 |
| 2.2 | Edge-adequate test suite for `avg`. | 7 |
| 2.3 | All-uses-adequate tests with respect to variable *sum* in `avg`. | 8 |
| 2.4 | Grammar for formulas. | 12 |
| 4.1 | Node- and edge-adequate test suite for `rootsolver`. | 29 |
| 4.2 | Cell-dependence-adequate test suite for `rootsolver`. | 31 |
| 4.3 | Du-association adequate test suite for `rootsolver`. | 33 |
| 4.4 | Du-validation adequate test suite for `rootsolver`. | 37 |
| 5.1 | Data about experimental subjects. | 53 |
| 5.2 | Nonexecutable du-associations. | 59 |

WHAT YOU SEE IS WHAT YOU TEST
A TESTING METHODOLOGY FOR FORM-BASED VISUAL PROGRAMS

Chapter 1

# INTRODUCTION

Visual programming languages employ visual representation as a means of programming to accomplish tasks that might otherwise have to be encoded in a traditional one-dimensional programming language [33]. One important class of visual programming languages — form-based visual programming languages — provides a declarative approach to programming, characterized by a dependence-driven, direct-manipulation working model [1]. 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 database access [39], for providing steerable simulation environments for scientists [6], and for supporting the specification and implementation of full-featured GUIs [23].

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 [4], 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 [8]. Compounding this problem, creators of spreadsheets

express unwarranted confidence in the reliability of their programs [4]. 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, 16, 20, 30, 38]) is directed at imperative programs.

In this paper,* we first discuss various issues that impact testing strategies for 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.

An important issue for testing involves test adequacy criteria, which provide a means for assessing how well we have tested a program. We define several test adequacy criteria for form-based programs, and illustrate their application. We then show that a criterion analogous to the traditional "all-uses" dataflow test adequacy criterion is particularly appropriate for form-based programs, because it exercises interactions both between and within cells.

Based on these results, we present 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 variant of the all-uses 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 form-based languages, our methodology includes an interface 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. Thus our methodology is appropriate for use by a wide range of programmers, including the many end users who use spreadsheets.

---

* Much of the material contained in this thesis has appeared previously in [31] and [32].

Finally, we present the results of an empirical study that we have performed to investigate the effectiveness of our methodology in detecting faults in form-based programs. Our empirical results are encouraging. The overall average (mean) percentage of faults detected for all programs, faulty versions, and test suites in our study was 81%, which suggest that in practice, our methodology can achieve fault detection results at least comparable to the results achieved by analogous techniques for testing imperative programs. These results imply that the potential benefits of our approach to users of form-based programming languages may be substantial.

# Chapter 2

# BACKGROUND

## 2.1 Software testing

Software testing is the process of dynamically executing a program or program component for some chosen input, and verifying that the program produces the correct output for that input. Let P be a program we wish to test; to judge whether P's output is correct, we assume there is a specification S for P which tells us, for each test input, whether P's output is correct. In practice, S may or may not physically exist. For convenience, we assume the existence of an *oracle* that tells us whether output is correct. Some consequences of this assumption are discussed in [34].

Though testing involves validating output, testing itself cannot in general ensure program correctness because many programs have infinite input domains, or input domains that are too large to exhaustively test. Normally, a subset of an input domain must be chosen.

This leads to some general problems, for example, how many tests do we need to test a given program? How do we choose those tests? When should we consider that our testing is adequate and stop testing? These problems are very important to research and industrial practice. Testing methodologies address these problems.

Two prominent classes of testing methodologies are *specification-based* (*functional, black-box*) testing and *code-based* (*structural, white-box*) testing. Specification-based testing selects test data without implementation knowledge of the program, whereas code-based testing uses explicit knowledge of program structure to guide

test data selection. In this paper, for reasons explained later, we focus on code-based testing.

Test adequacy criteria provide a way to select test data, and decide when to stop selecting data. Adequacy criteria are based on the premise that certain aspects of the code or specifications must be "exercised" or "covered". Code-based adequacy criteria require the tester to select and execute tests that exercise certain components of code.

Code-based test adequacy criteria are frequently defined on abstract models of programs rather than directly on code itself. For imperative programs, a frequently used abstract model is the *control flow graph*, which is a graphical representation of a procedure's control structure. One specific control flow graph is a directed graph in which each node represents a simple or conditional statement, and each edge represents the flow of control between statements. Two simple code-based test adequacy criteria based on control flow graphs 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 (i.e., may represent statements or paths of statements that cannot be executed for any input to the procedure), these criteria are rendered *applicable* by restricting coverage requirements to executable program components [11].

Figure 2.1 shows program avg and its control flow graph. Program avg takes an array $a$ containing up to ten integers, and a variable *count* that says how many integers are actually stored in $a$ on this call, and returns the average of those integers in $a$, or returns $result = -1$ if *count* is out of range. In the control flow graph for avg, statement nodes, shown as ellipses, represent simple statements. Predicate nodes, shown as rectangles, represent conditional statements; labeled edges out of these nodes represent control paths taken when the conditional statement evaluates

to the value of the edge label. For reference, we label each node in the graph with the number of the corresponding statement in original program. Nodes E and X represent entry to, and exit from, the program, respectively.

```
avg(a: array[1..10] of int,count:int,result:int)
    int i, sum
    begin
1.      result = -1;
2.      if(count >=0 AND count <=10)
3.          sum = 0
4.          i = count
5.          while(i > 0) do
6.              sum = sum + a[i]
7.              i = i - 1
            end
8.          result = sum/count
        end
    end
```

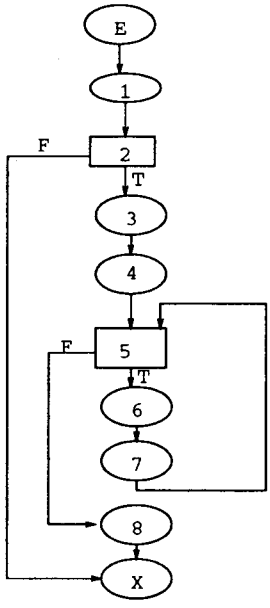


FIGURE 2.1: Control flow graph of program avg.

Node and edge adequacy criteria are based directly on control flow. Table 2.1 illustrates a node-adequate test suite for avg and Table 2.2 illustrates an edge-adequate test suite for avg. Notice that a single test exercises all nodes in avg, whereas at least two tests are required to exercise all edges.

Control-flow based testing techniques are often combined with data flow analysis, which calculates interactions between variable definitions, and places where defined values are used. Test adequacy criteria based on these interactions are called dataflow test adequacy criteria. Many such adequacy criteria have been proposed for use

| Test | Inputs | Node coverage |
|---|---|---|
| 1 | {a=10,2,4,6; count=4} | {E,1,2,3,4,5,6,7,8,X} |

TABLE 2.1: Node-adequate test suite for avg.

| Test | Inputs | Edge coverage |
|---|---|---|
| 1 | {a=10,2,4,6; count = 4} | (E,1),(1,2),(2,3),(3,4),(4,5), (5,6),(6,7),(7,5),(5,8),(8,X) |
| 2 | {a=10,2,4,6; count=-2} | (E,1), (1,2), (2,X) |

TABLE 2.2: Edge-adequate test suite for avg.

with imperative programs(e.g.,[20, 24, 28]). Dataflow criteria examine *definition-use associations* (*du-associations*), which are pairs consisting of a definition and a use of a variable, such that there is a control flow path from the definition to the use on which there is no intermediate redefinition or undefinition of the variable. A *definition* is an operation that defines a variable's value. A *use* is an operation that reads a variable's current value. There are two types of uses. *C-use* directly affects the computation being performed and may indirectly affect the flow of control through the program. The second type of use, a *p-use*, occurs in a predicate statement where it directly affects the flow of control, and may indirectly affect the computation. Dataflow-based adequacy criteria stipulate that a test suite must exercise certain du-associations. By varying the required combinations of definitions and uses, a family of test data selection and adequacy criteria is defined in [17, 28]. In this work,

we restrict our attention to the *all-uses* criterion, which requires that test data cause the traversal of at least one path from each variable definition to every p-use and every c-use of that definition.

Table 2.3 illustrates an all-uses adequate test suite with respect to variable *sum* in program avg. In the table, du-associations are listed in format (definition, use); definition and use are listed in format n:x where n is the node that contains the definition or use, and x is the defined/used variable. For example, (3:sum, 6:sum) means that *sum* is defined in statement 3 and used in statement 6.

| Test | Inputs | du-associations exercised |
|---|---|---|
| 1 | {a=10,2,4,6; count = 4} | (3:sum, 6:sum) (6:sum, 6:sum), (6:sum, 8:sum) |
| 2 | {a=10,2,4,6; count = 0} | (3:sum, 8:sum) |

TABLE 2.3: All-uses-adequate tests with respect to variable *sum* in avg.

## 2.2 Form-based visual programming languages

### *2.2.1 The form-based programming language paradigm*

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, and users fill in those blanks by entering (constant) formulas. The best-known examples of form-based languages are commercial spreadsheets (see Figure 2.2). But there are also many other 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 [19].

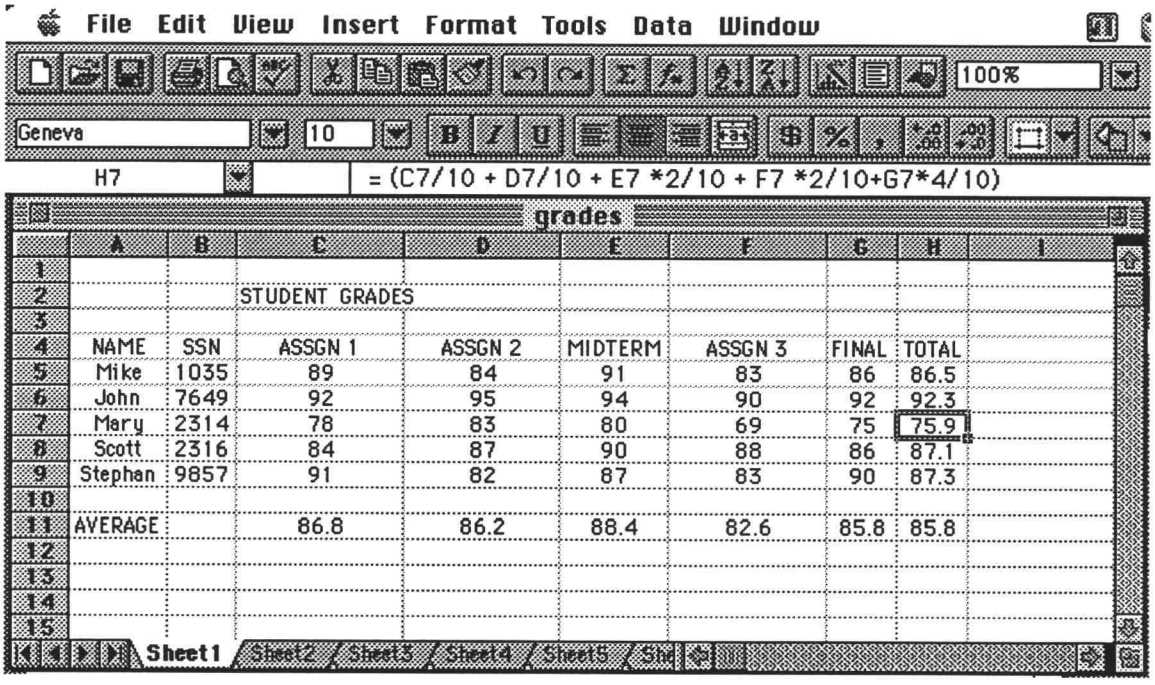


FIGURE 2.2: Spreadsheet to calculate student grades.

### 2.2.2 *Forms/3: a form-based visual programming language*

Forms/3 is another form-based programming language [5, 12]. We use one example to show how a user would construct a graphical clock in Forms/3. Figure 2.3 shows each cell with its formula.

We use the term *input cell* to mean cells whose formulas contain only constants. `Clock` consists of 13 cells, including two input cells (upper left) that could eventu-

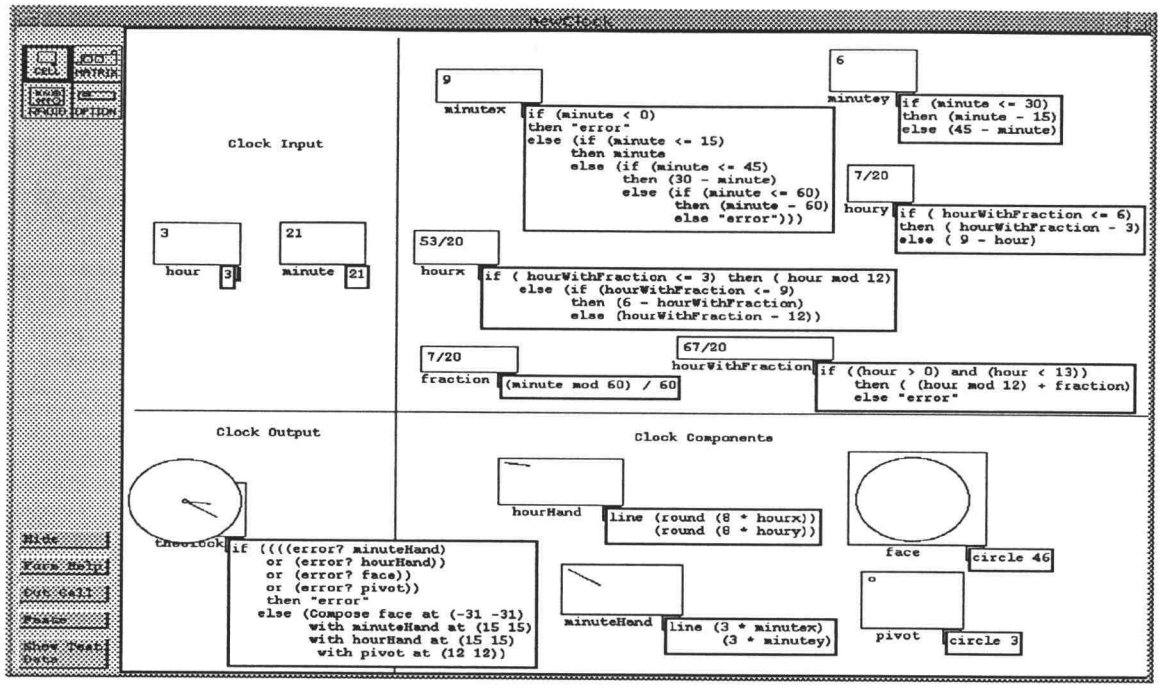


FIGURE 2.3: Programming a clock in Forms/3.

ally be replaced with references to the system clock, one output cell (middle left), and several cells used in intermediate calculations (sections at right). 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 Figure 2.4.

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 2.4. The Forms/3 figures presented in this paper were programmed using this subset. From this grammar, it is clear that

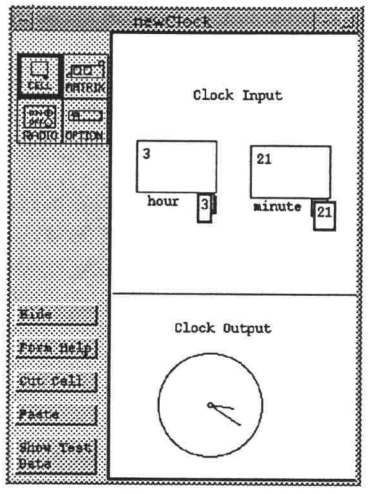


FIGURE 2.4: User's view of program `clock`.

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.

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

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

```
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 2.4: Grammar for formulas.

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.

### 2.2.3 *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.

## 2.3 Related work

As mentioned earlier, much research on testing imperative programs has been done. However, our search of the research literature has revealed no published research on the topic of testing form-based visual programs. Some recent research [2, 3, 21], however, has addressed problems of testing for logic programs written in Prolog. Because Prolog programs and form-based visual programs are both declarative in nature, it is reasonable to ask whether they can be tested similarly.

To support control-flow based coverage testing of Prolog programs, Luo et al. [21] present two types of graphs that explicitly represent the hidden control flow of Prolog programs. They compare Prolog programs with imperative programs; Prolog programs are recursive in nature (their basic data structures are recursive lists) and important differences between flow of control in Prolog and in imperative programs stem from the fact that subgoal unification is bidirectional, and backtracking occurs after failure. Luo et al. define control flow graphs that accommodate these differences, and based on these control flow graphs, define test selection criteria analogous to criteria defined for imperative programs.

Azem, Belli et al.[2] investigate implementation-based testing and reliability determination of logic programs, focusing on Prolog. They describe a testing environment, PROTest, and a reliability assessment environment, PRORool. In subsequent work, Belli and Jack [3] investigate test coverage measures, working from a different view than Luo et al. of the implicit control flow in a Prolog program. Instead of attempting to extract a control flow graph from a Prolog program, they define an abstract model of the program, which is a set of goal-induced instances of program clauses, and define on the model a coverage measure by the least upper bound of the clause instances.

Logic languages and form-based visual languages are both declarative in nature: neither language encodes explicit control flow beyond that which can be said to occur within formulas. Both classes of languages depend on an engine to determine dependencies and evaluate programs. However, the evaluation schemes behind the two types of languages are quite different. Logic programming is based on first-order predicate logic; a Prolog program consists of a finite set of definite Horn clauses. In Prolog, the basic data structure is a recursive list, and unification and backtracking are the mechanisms that drive computations. When goal unification is complete, a solution is given. Unification and backtracking, however, cause control flow and data flow to be bi-directional. The differences between the paradigms motivate different approaches to testing and defining coverage criteria for the two types of languages.

# Chapter 3

# ISSUES AND OPPORTUNITIES

Form-based visual programs are subject to the same overall concerns, with respect to testing, as imperative programs; however, differences between the form-based and imperative language paradigms influence attempts to develop testing methodologies for form-based programs.

In this section we describe several differences between the imperative and form-based language paradigms, and discuss the 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 [4] 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

One important difference between form-based visual languages and traditional imperative languages involves their user base. A large part of the form-based visual language user base includes "end user" programmers such as business professionals, or scientists doing exploratory work, few of whom are likely to create specifications for their programs. Moreover, code of form-based visual program is more accessible than that of imperative textual program. 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.

## 3.3 Incremental versus nonincremental testing

The form-based language paradigm supports testing processes that differ from processes readily supported by the imperative paradigm.

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 [1]. 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.

Applied to form-based programs, incremental and nonincremental testing processes achieve the same overall results: they both test the application of entire input vectors to programs. To illustrate, suppose form-based program $P$ has input cells $C_1$, $C_2$, and $C_3$, and we wish to run two tests on $P$: test 1 applies input vector $(i_1, i_2, i_3)$, and test 2 applies input vector $(i_1, i_2, i_4)$. Using the nonincremental approach we enter all test 1 inputs, then validate output cell contents, then enter all test 2 inputs, and then validate output cell contents. With the incremental approach, we enter all test 1 inputs, then validate output cell contents, then achieve test 2 setup by altering $C_3$ to $i_4$, and then validate output cell contents. For both of these approaches, the end result is the application and validation of the two input vectors.

The approaches differ, however, in two ways. The incremental approach reduces the effort required to test $P$, by reducing the number of inputs that the tester must enter. Furthermore, when we apply test 2 incrementally, we need only validate output cells that are affected by modified input $(i_4)$; outputs that depend only on inputs entered previously $(i_1, i_2)$ 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 programs 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, 27]. 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 language level, given program $P$, an evaluation engine can evaluate $P$'s cells in any order that preserves data dependencies. 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 a tester to select test data that exercises 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 $P$. Given 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 $P$ [35]. 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 [36]; 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 part 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 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, 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. Testing strategies for form-based programs can utilize this responsiveness to integrate development, testing, and debugging activities. 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. One concept that is not foreign to form-based language programmers is the concept of cells interacting through cell references [4]. 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.

Chapter 4

# A METHODOLOGY FOR TESTING FORM-BASED VISUAL PROGRAMS

In Chapter 3, we discussed differences between the form-based visual language paradigm and traditional imperative paradigms. Some of these differences – such as the incremental, responsive nature of programming environments for form-based languages, and the need to support users unschooled in formal notions of testing – add to the difficulty of creating a practical testing methodology. But also, these differences provides opportunities for integrating testing, debugging, and validation efforts to an extent not found with traditional imperative programs.

In this chapter, we present a methodology for testing form-based visual programs. We organize the material into two sections: in Section 4.1 we describe a family of test adequacy criteria for form-based visual programs, and compare and contrast these criteria analytically. We show that a criterion based on the all-uses dataflow adequacy criterion is particularly appropriate for form-based visual programs. In Section 4.2 we describe our methodology for testing form-based visual programs, that utilizes this criterion in a way that takes into consideration the differences between form-based visual programs and imperative programs.

## 4.1 Adequacy criteria for form-based programs

We define five test adequacy criteria. The first two criteria focus solely on formulas. By failing to consider dependencies between cells, however, these criteria can miss

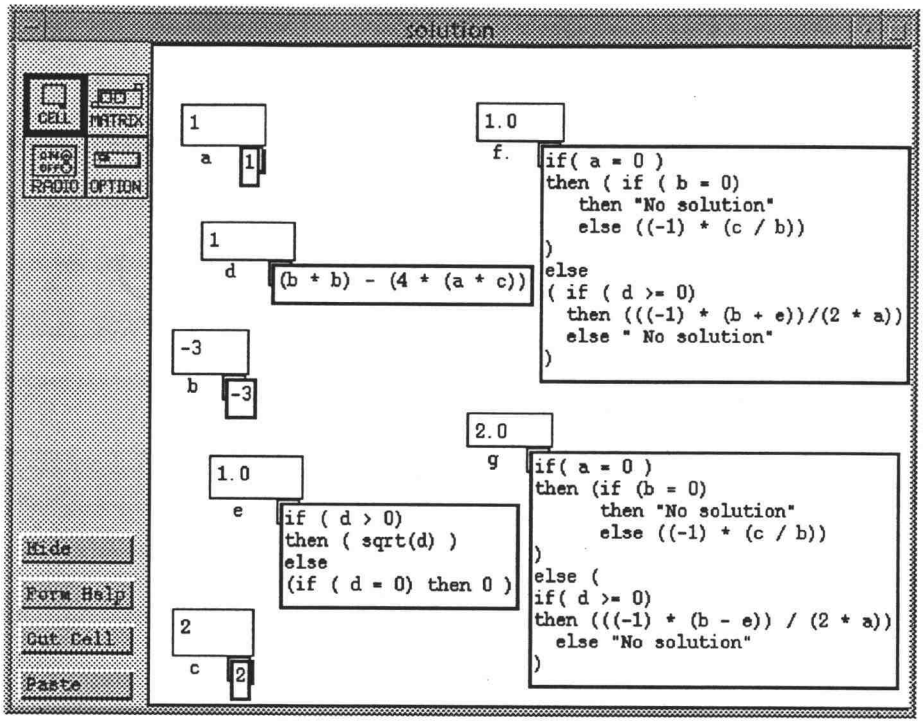


FIGURE 4.1: Forms/3 program to obtain the root(s) of quadratic or linear equations.

significant interactions. Our third criterion is based on simple dependencies between cells. We show that this criterion is weaker than the first two criteria, in part because it does not consider the effects of control flow within cell formulas on dependencies between cells. Next, we describe a data-flow adequacy criterion that overcomes the deficiencies of the first three criteria: it forces testers to exercise dependencies between cells, but also to consider the effects of control flow within cell formulas on those dependencies. Finally, we describe a du-validation criterion, which requires the same coverage as the all-uses criterion, but determines coverage with respect to validated outputs.

As a running example, we use the Forms/3 program `rootsolver`, displayed in Figure 4.1, that calculates the solutions of the equations $ax^2+bx+c = 0$ or $bx+c = 0$. 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 $bx + c = 0$.

### *4.1.1 An abstract model for form-based programs*

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 and discussed in Section 2.1. 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 4.2 displays the formula graph for cell `f` of `rootsolver`, using notation similiar to that used for the control flow graph in Figure 2.1. For reference, we label each node in the graph with the name of the associated cell and an integer.

To model dependencies between cells, we represent them as edges between formula graphs. 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 $A$. We calculate cell dependencies by analyzing cell formulas

and locating references, in each formula, to other cells. 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 4.3 displays the CRG for `rootsolver`. In the figure, formula graphs

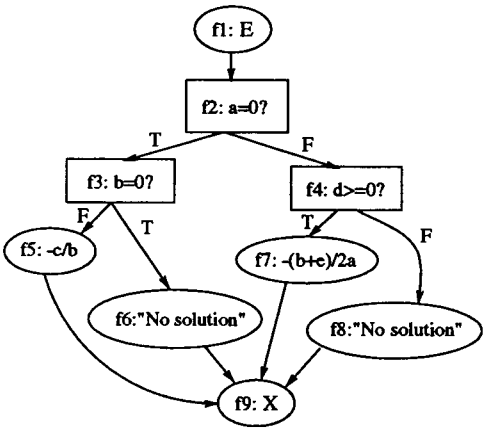


FIGURE 4.2: Formula graph for cell `f` of `rootsolver`.

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. The presence of a cell dependence edge between cells $A$ and $B$ in the CRG for program $P$ means only that when $P$ is executed, the evaluation engine must execute cells in $P$ in an order that preserves this dependence. If $(a_x,b_e)$ is a cell dependence edge from cell $A$ to cell $B$ 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

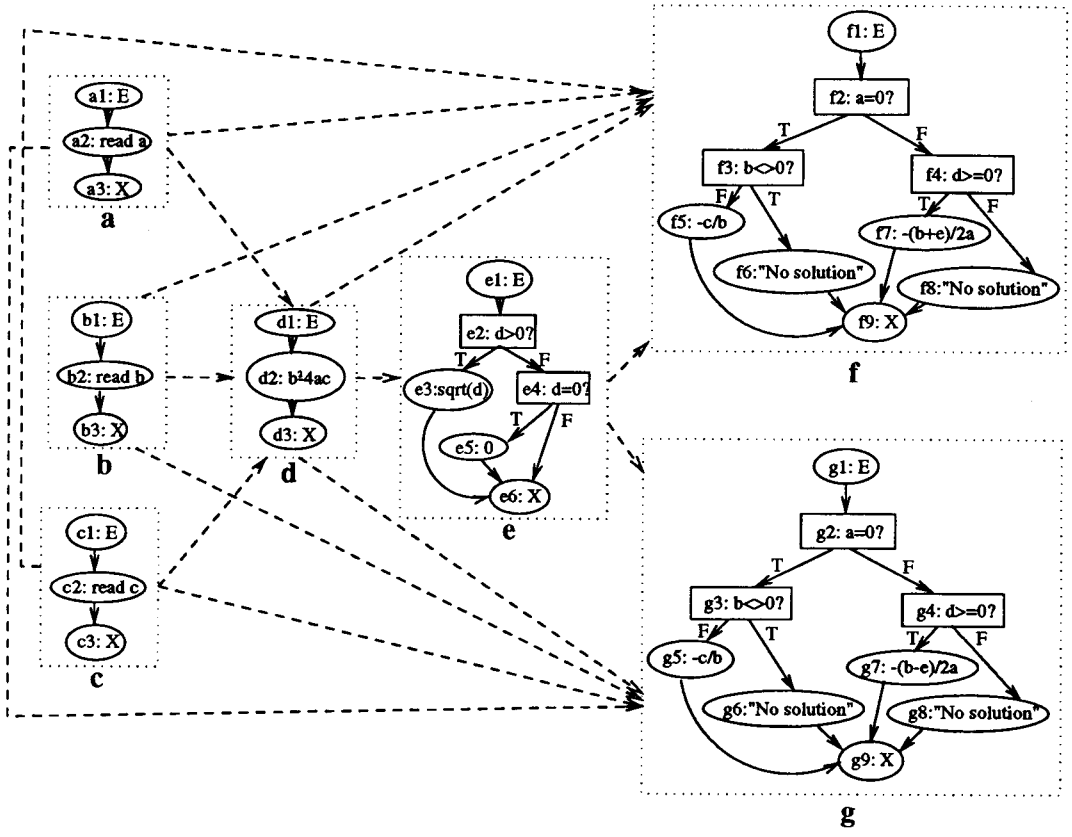


FIGURE 4.3: Cell relation graph for `rootsolver`.

on the underlying evaluation scheme and on other dependencies in $P$. In contrast, edges within formula graphs do represent control flow. 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.

### *4.1.2 Node and edge adequacy criteria*

We use formula graph sets to define analogues, for form-based programs, of the applicable node and edge adequacy criteria utilized for imperative programs and defined in Section 2.1. 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 4.1 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 nodes and edges in each of those cells; however, no single test case exercises all coverable components. 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 4.1 lists the nodes and edges exercised.

The test suite shown in Table 4.1 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 ($e4$,$e6$). In this sense, the edge-adequate criterion is stronger than the node-adequate criterion: we can satisfy the node-adequate criterion without testing some potentially important decisions.

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

TABLE 4.1: Node- and edge-adequate test suite for `rootsolver`.

### *4.1.3 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 4.1 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 $B$, 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 $(x, y)$.

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 4.1. When we remove test cases 3 and 4 from that test suite, the resulting test suite, shown with cell dependence coverage information in Table 4.2, 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),<br>(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),<br>(c,f),(a,g),(b,g),(c,g) |

TABLE 4.2: Cell-dependence-adequate test suite for `rootsolver`.

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.

As discussed in Section 2.1, 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. However, 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 form-based program $P$, with formula $F$ and formula graph $\bar{F}$. Each node in $\bar{F}$ that represents an expression that assigns a value to $C$ is a *definition* of $C$. Each non-predicate node in $\bar{F}$ that represents an expression referring to cell $D$ is a *c-use* (computation use) of $D$. 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 as-*

*sociation* 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. Note that these definitions specify only *executable* du-associations: associations for which there exists some input for which the definition reaches the use.

To utilize definition-use associations for test adequacy purposes, we must define what it means to exercise an association. Given du-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 an evaluation of the formula that contains the definition, that evaluation corresponds to a traversal of a path through the formula graph (for C) that contains n1, and then, later in the computation, $t$ causes evaluation of $F$, and that evaluation traverses a path through $\bar{F}$ that contains $n_2$ (or $(n_2, n_3)$).

For imperative programs, a wide range of dataflow test adequacy criteria have been defined; each yields a particular coverage of du-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 [24]. 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 du-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 du-association

| Test | Inputs | du-association coverage |
|---|---|---|
| 1 | {a=1, b=-3, c=2} | (a2,d2,a),(b2,d2,b),(c2,d2,c),(d2,(e2,e3),d), (d2,e3,d)<br>(a2,(f2,f4),a),(a2, f7,a),(d2,(f4,f7),d),(b2,f7,b),(e3, f7,e)<br>(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)<br>(d2,e3,d),(a2,(f2,f3),a),(a2,(g2,g3),a),(b2,(f3,f5),b)<br>(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)<br>(d2,(e2,e4),d),(d2,(e4,e5),d),(a2,(f2,f3),a)<br>(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)<br>(d2,(e2,e4),d),(d2,(e4,e6),d),(a2,(f2,f4),a)<br>(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),<br>(a2,(f2,f4),a),(d2,(f4,f7),d),(a2,(g2,g4),a),(d2,(g4,g7),d)<br>(a2,f7,a),(b2,f7,b),(e5,f7,e),(a2,g7,a),(b2,g7,b),(e5,g7,e) |

TABLE 4.3: Du-association adequate test suite for `rootsolver`.

does not involve paths between cells that contain definitions and cells that contain uses, the criterion simply requires test data to "exercise each executable association."

Table 4.3 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 4.1.3.

The superiority of all-uses adequacy illustrated in this example reflects a relationship that also has been suggested to exist in the imperative-language counterparts of these criteria — a relationship that has been examined both analytically and empirically (e.g., [11, 17]). 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.

Although we do not present them fully here, we can also define analogues of other dataflow adequacy criteria for form-based programs. However, not all of these criteria appear particularly useful for this application. For example, an *all definitions* criteria could require each definition that has at least one use to reach at least one such use; however, dependence-driven evaluation ensures that, if a test suite executes all nodes, it will also execute each such definition to at least one use. On the other

side of the issue, a criterion that might be useful for form-based visual programs is the *required k-tuples* criteria [24], which requires execution of k-length chains of du-associations. Consideration of all criteria such as these is a subject for future research.

### *4.1.4 The du-validation criterion*

Our all-uses dataflow adequacy criterion is particularly appropriate for form-based programs. As Duesterwald, Gupta, and Soffa [9] observe, however, merely exercising a du-association does not necessarily demonstrate its correctness. Typically, an incorrect du-association is detected by the programmer only if its erroneous effect is reflected in some computed output value. The importance of connecting du-associations to validated outputs is particularly clear for form-based programs, because in these programs, cells may be hidden or be off the screen, and inputs may exercise many du-associations that do not lead to visible (and thus, verifiable) outputs.

Thus, we define a *du-validation criterion* for use on form-based programs, that relates test adequacy to validated outputs. The du-validation criterion, like the all-uses criterion, requires that all du-associations in the program be exercised by some test; thus, the two criteria are equivalent in terms of the program components that they require a tester to exercise.

However, for the du-validation criterion, we modify our definition of a test so that a test includes both an assignment of values to a program's input cells, and a designated output cell. We consider a du-association *dua* to be exercised by a test only if, when the program is run for that test's input, *dua* contributes (directly or transitively) to the computation of the final value of the designated output cell.

This validation-driven test adequacy criterion is well suited for incremental testing of form-based programs, given their responsiveness. The user may apply a test by entering an assignment of values to input cells — either by entering new values in all cells or entering values only in cells that must be changed in order to reach the input assignment required by the test. Then, the user pronounces an output cell's value "valid" for this test input. The du-association that contribute directly or indirectly to this output value can then be calculated, by a process we describe in the next section.

Table 4.4 displays a du-validation-adequate test suite for `rootsolver`. As the table indicates, the test suite exercises the same du-associations as the all-uses-adequate test suite depicted in Table 4.3.

## 4.2 An incremental and validation-driven methodology

To provide a variety of users some benefits of formal testing without requiring their understanding of formal testing methodologies, we developed a testing methodology. To accommodate differences between the form-based visual language paradigm and traditional imperative paradigms, our testing methodology 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 the dynamic du-associations that currently influence calculations, using a "probe" inserted into the evaluation engine.

- A user-accessible facility for pronouncing outputs "validated" at any point during program development, and the abilities both to determine the du-

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

TABLE 4.4: Du-validation adequate test suite for `rootsolver`.

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 will discuss in detail how our methodology provides these functionalities to form-based languages. We present the material in the context of a prototype implemented within the Forms/3 programming environment. More detail about the implementation is presented in Section 5.1 and the Appendix. Although this prototype is for Forms/3, and examples are presented with Forms/3, the methodology is not Forms/3-specific; it could be implemented, with appropriate substitutions for user interface components, for other form-based visual programming languages. We will use the example program *Clock*, discussed in Section 2.2, to illustrate our testing methodology. Figure 4.4 shows *Clock*'s CRG.

### *4.2.1 Task1: Collecting static du-associations.*

Suppose that, starting with an empty form, the user begins to build the `Clock` application by entering cells and formulas, reaching the state shown in Figure 4.5. Assume for the moment that the user does not change any formulas, but simply continues to add new ones. (We remove this restriction later.)

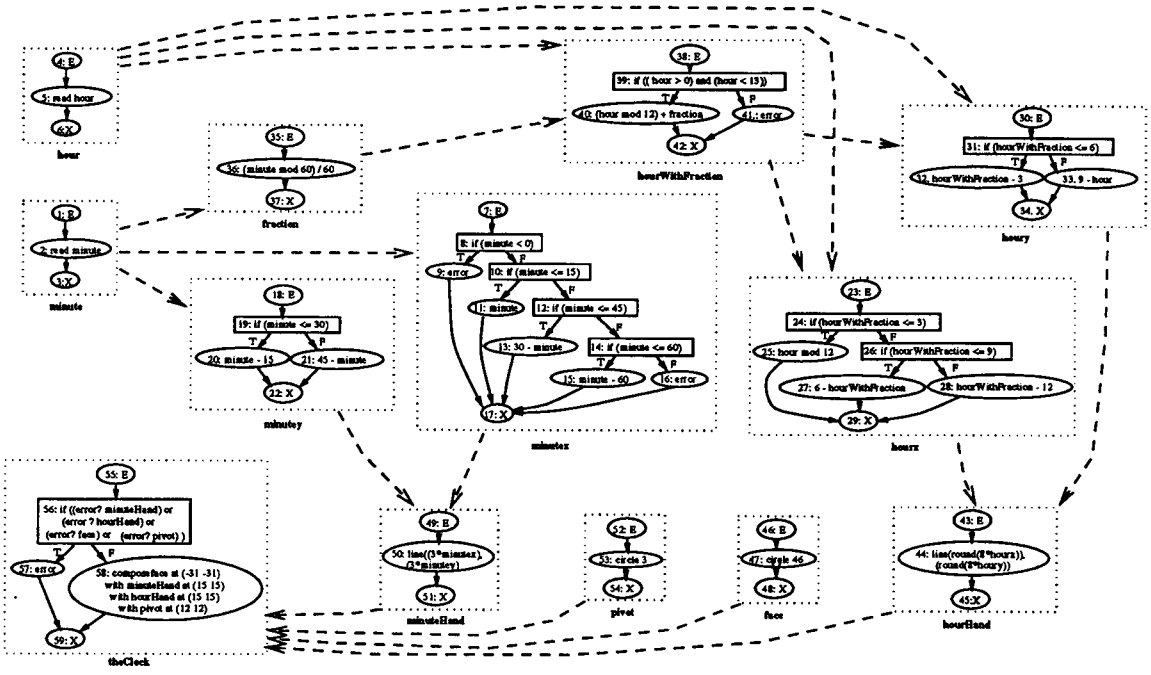

FIGURE 4.4: `Clock`'s cell relation graph.

Because it would be expensive to exhaustively compute the du-associations for the entire program after each new formula is added, we compute them incrementally. Several algorithms for incremental computation of data dependencies exist for imperative programs (e.g., [22, 26]), 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

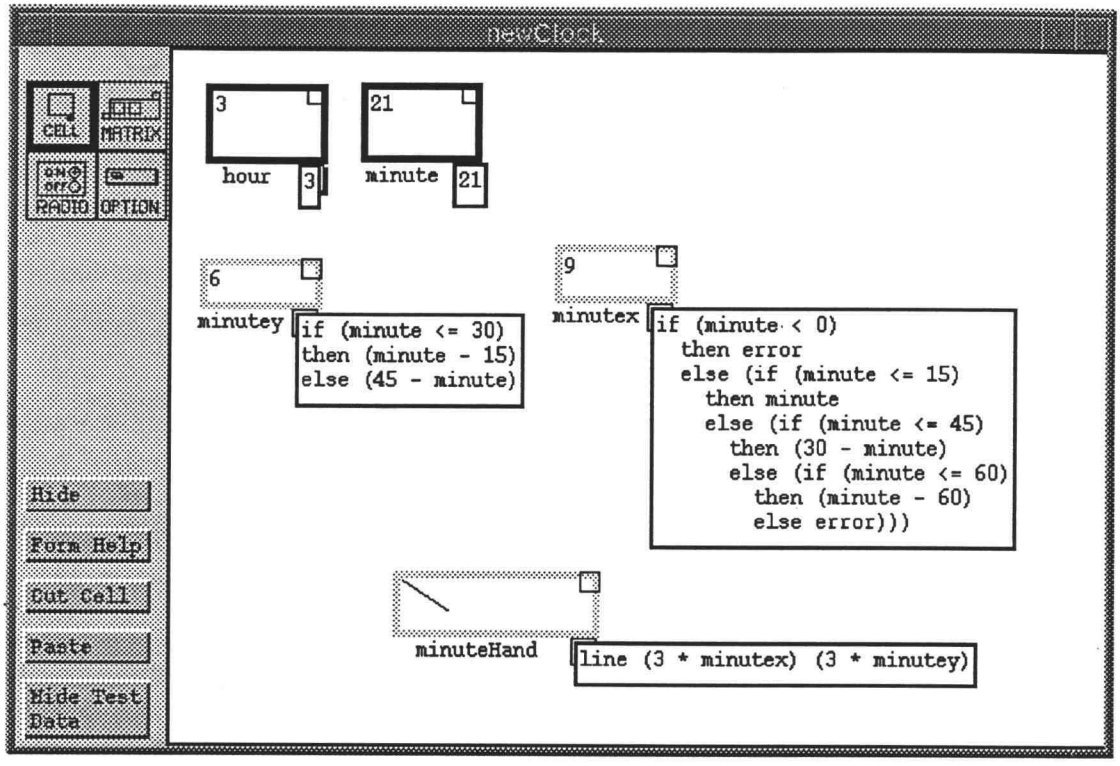


FIGURE 4.5: `Clock` at an early stage; part of the program has been entered.

new cell and all cells that are directly referenced by the new cell.* 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 $C$.

We can use 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 $C$ ref-

* 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, and virtually all form-based languages use it to varying extents.

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

FIGURE 4.6: Algorithm for collecting du-associations.

erences; $C.LocalDefs$, 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 4.6 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 4.5, suppose that the most recent formula entered is that for cell `minuteY`. 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 4.4 as: (2,(19,20),`minute`), (2,(19,21),`minute`), (2,20,`minute`), (2,21,`minute`), (20,50,`minutey`), and (21,50,`minutey`).

`CollectAssoc` runs in time $O(udn)$, 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`.

### *4.2.2 Task 2: Tracking dynamic du-associations.*

To incrementally track du-associations that have been exercised, we simply insert a probe into the evaluation engine. When cell $C$ executes, this probe records the execution trace on $C$'s formula graph, storing it in $C.Trace$. For example, in the case of `Clock`, at the moment depicted in Figure 4.5, the execution trace stored for cell `minutey`, described in terms of Figure 4.4's node numbers, is (18,19,20,22). 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.

### *4.2.3 Task 3: Pronouncing outputs "validated".*

In this section, we show how the data collected in Tasks 1 and 2 can 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 4.7 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 automatically changes to a question mark (not shown in the figure), which means that 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 have been exercised). (In this black-and-white paper, the continuum

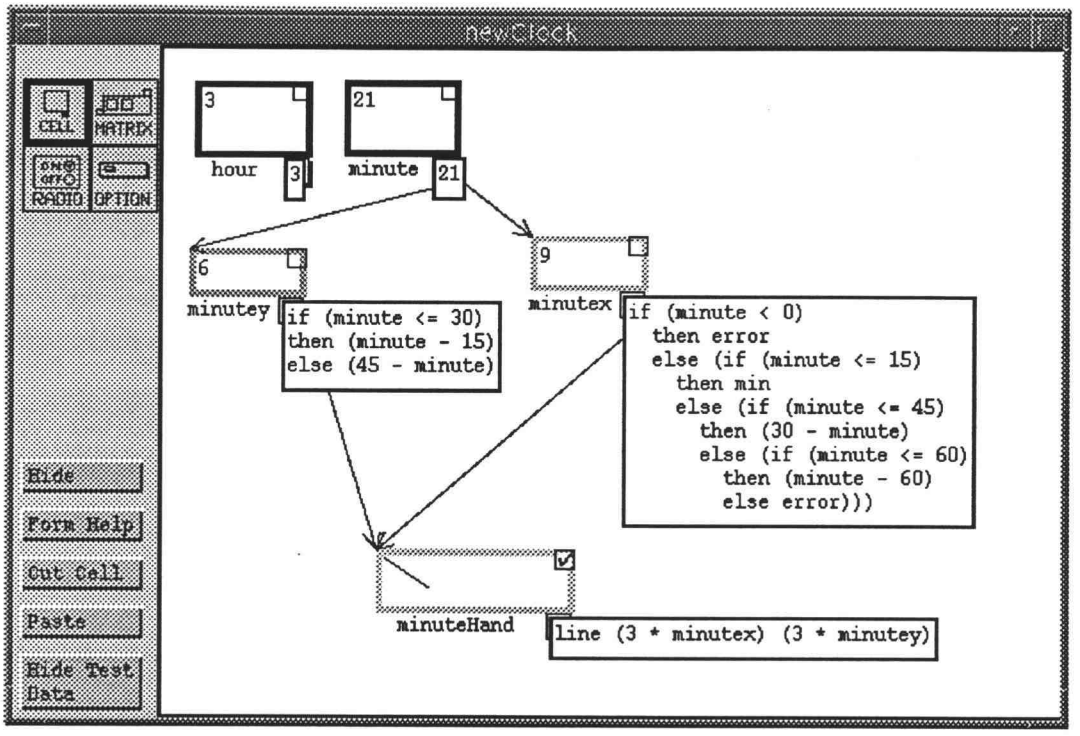


FIGURE 4.7: The evolving `Clock` program at an early stage, after the `minuteHand` cell has been validated.

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 4.7, 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; to provide better information, we could implement 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 4.8 displays our algorithm `Validate`, which is invoked when the user pronounces a displayed value valid. The algorithm uses information about static and dynamic du-associations, previously calculated and stored as discussed earlier, to determine which du-associations participate in the production of $C$'s current value, and to update $C$'s border. The algorithm then recursively does the same for each referenced cell that has contributed to the computation.[†]

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, it is not masked by the cost of the evaluation process. $ValidatedID$ is set to 0 when the programming environment is first activated. 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.[‡]

---

[†] A generalization of this algorithm related to the approach of [9] uses slicing algorithms 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 functions for programs with recursion, iteration, and redefinitions of variables. For most form-based languages, however, the more efficient `Validate` approach suffices.

[‡] 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.

```
1.  algorithm Validate(C)
2.  ValidatedID = ValidatedID + 1
3.  C.ValTab = "checkmark"
4.  ValidateCell(C)

5.  procedure ValidateCell(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.         ValidateCell(D)
13. UpdateBorder(C)
```

FIGURE 4.8: Algorithm for updating test adequacy information following a validation request.

The running time of `Validate` is bounded by the number of calls to `ValidateCell`, times the cost of each call. The number of calls to `ValidateCell` is bounded in the worst-case by the number of cells in the program, but is proportional to the number of cells that contribute to the current computation of the validated cell as encoded in dynamic execution traces, which in many practical form-based programs is much smaller than the total number of cells in the program. The cost of a call to `ValidateCell` is bounded by the number of uses that appear in a cell's execution trace; although in the worst case this may equal the number of cells in the program it too is typically bounded by a small integer.

### 4.2.4 *Task 4: Adjusting test adequacy information.*

So far, we have focused on how form-based programming environments can handle cell formulas as they are added to a form-based program. We now consider the other

We assume that $ValidatedID$ will not overflow, to simplify the presentation.

basic edits possible with form-based programs, namely, deleting a cell or changing a cell's formula. Changes to an input cell's formula have already been handled, and 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. This lack of awareness about the potential effects of changes may be an important factor in the overconfidence users exhibit about their form-based programs.

To address this lack of awareness, the system must immediately reflect the new test adequacy status of the program whenever a cell is modified.[§] To accomplish this, the system must (1) update $C$'s static and dynamic du-associations, 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.

[§] 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., [22, 26]) and incremental dataflow testing (e.g. [14, 15, 29] 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 or redefinitions of variables. For most form-based languages, however, the simpler approach that we present here suffices.

Our methodology handles item (2) first, removing the old information before adding the new. Let $C$ be the modified cell. We use a conservative approach that recursively visits affected cells. The algorithm, `UnValidate`, given in Figure 4.9, 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 and dynamic du-associations 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 Section 4.2.1 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.

### *4.2.5 Task 5: Batch computation of information.*

Test information can be saved when a program is saved; then, when the program is reloaded for further development, it is not necessary to exhaustively reanalyze it. There are some circumstances, however, in which it may be necessary to calculate

```
1.  algorithm UnValidate(C)
2.  AffCells = {}
3.  UnValidatedID = UnValidatedID + 1
4.  UnValidateCell(C)
5.  for each cell D ∈ AffCells do
6.      UpdateBorder(D)
7.      UpdateValTab(D)

8.  procedure UnValidateCell(C)
9.  C.UnValidatedID = UnValidatedID
10. AffCells = AffCells ∪ C
11. for each cell D ∈ C.CellsThatRef do
12.   for each definition d (of C) in C do
13.     for each ((d,u),true) ∈ D.DUA do
14.       D.DUA = D.DUA∪{((d,u),false)}−{((d,u),true)}
15.   if D.UnValidatedID < UnValidatedID then
16.     UnValidateCell(D)
```

FIGURE 4.9: Algorithm for updating test adequacy information following a modification.

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, a more efficient algorithm takes an entire set of cells as input, and makes passes over this set to update information on du-associations and validation status.

# Chapter 5

# IMPLEMENTATION AND EMPIRICAL STUDY

## 5.1 Implementation

As mentioned and illustrated in the previous chapter, we have implemented a prototype of our testing methodology within the Forms/3 programming environment. Our prototype incorporates all of the algorithms described in this paper except for that of Figure 4.9.* The screen shots used in this paper are from this prototype. This prototype is useful as a means for illustrating our testing methodology, but more importantly, it facilitates empirical studies of the methodology. In this chapter, we briefly overview the implementation of our prototype, then, we discuss our empirical study and results.

### *5.1.1 General environment*

The Forms/3 programming language and environment runs on UNIX, and is implemented in Lucid Common LISP; its user interface is implemented using the GARNET graphical toolkit [13], which supplies an object-oriented graphics system and a set of techniques to specify an object's interactive behavior in response to input devices. Our implementation is integrated within this LISP and GARNET environment.

---

* The author implemented all algorithms; however, the user interface for this prototype was implemented by Christopher Dupuis.

### 5.1.2 *Primary data structures*

In Section 4.2, we described the main functionalities of our testing methodology. Two primary requirements to implement the functionalities are the ability to calculate static du-associations and the ability to track dynamic du-associations.

Our implementation utilizes two basic data structures, DefTable and DUpairTable, to keep and update both static and dynamic du-association information. These structures are implemented as hash tables because hash tables support efficent searching and because our LISP version provides hash tables as a basic data type.

Each cell has an entry in DefTable, in which the cell ID serves as key. Each table entry is associated with a structure containing two fields: static-definition and dynamic-definition. A static definition is set whenever a new formula is added or a formula is modified. A dynamic definition is updated whenever this cell's formula is evaluated. A probe is inserted into the Forms/3 evaluation engine to track dynamic definitions.

Using DefTable, du-associations can be calculated by pairing each cell's definitions with each place the cell is referenced. Each du-association has an entry in DUpairTable, in which the du-association serves as key. Each table entry is associated with a status field. Initially all du-associations' statuses are set to "F". Whenever a user indicates that an output is "valid", the validation-backtracking algorithm sets the status of the du-associations that contributed to the validated output to "T".

### 5.1.3 *Main modules*

The main modules utilized in our prototype include the following:

- DefTable functionalities. Save all cells' static definitions and maintain their dynamic definitions.

- DUpairTable functionalities. Save all du-associations and maintain their status.
- Static du-association collection. Statically parse a cell's formula to determine all definitions for that cell.
- Dynamic du-association tracking. Dynamically track a cell's execution and keep that cell's current definition for that execution.
- Validation-backtracking. Pronounce a cell's output "valid", and from that cell, determine and mark all the du-associations that contributed to that cell's validated output.

Source code for main modules is listed in the Appendix.

## 5.2 Empirical study

We have used our prototype to perform an empirical study of the effectiveness of our technique at detecting faults. This is a fundamental topic of study; if our technique does not detect faults effectively in practice, there may be no reason to pursue it further.

### *5.2.1 Objectives*

The primary objective of our study was to measure the effectiveness of du-adequate test suites in detecting faults in form-based programs. A secondary objective was to investigate the degree to which form-based programs contain nonexecutable du-associations.

### *5.2.2 Experimental design*

The design of this study was partially patterned after the design of a study of imperative program testing reported in [17].

| *Program* | *Expr* | *DUA* | *Ver* | *Pool Size* | *Dtctn Ratio* | *Suite Size* |
|---|---|---|---|---|---|---|
| Clock | 33 | 64 | 7 | 250 | .24 | 11.3 |
| Digits | 35 | 89 | 10 | 230 | .09 | 22.7 |
| FitMachine | 33 | 121 | 11 | 367 | .18 | 30.2 |
| Grades | 61 | 55 | 10 | 80 | .10 | 9.8 |
| MicroGen | 16 | 31 | 10 | 170 | .09 | 10.4 |
| Sales | 30 | 28 | 9 | 176 | .17 | 10.4 |
| Solution | 20 | 32 | 11 | 99 | .12 | 12.0 |
| TimeCard | 33 | 92 | 8 | 240 | .12 | 16.7 |
| average | 33 | 64 | 10 | 202 | .14 | 15.4 |

TABLE 5.1: Data about experimental subjects.

For our study, we obtained eight Forms/3 programs. Table 5.1 lists details about these subjects. In the table, we include (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, fault detection ratio, and average size of du-adequate test suites for the program. 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).

To investigate the fault-detection effectiveness of our testing methodology, we asked seven users experienced with Forms/3 and/or 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.

The author, as a tester who had no knowledge of these specific faults, generated a pool of tests for each of the base versions of the subject programs. For each base program, the author first created tests of program functionality. Then the author 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. The author also verified that for all tests, validated cells in the base version produced correct values.

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. Because there may be a variety of inputs to a program by which a given du-association is exercised, distinct du-adequate test suites may differ in terms of their ability to reveal faults. Thus, for our study, we required a variety of du-validation-adequate test suites. We generated between 10 and 15 distinct du-validation-adequate test suites for each of our subject programs; Table 5.1 lists the average sizes of these test suites.

To measure the relative difficulty of detecting our faults, we determined their detection ratios. The *detection ratio* for a fault $f$ relative to test suite $T$ is the percentage of tests in $T$ that reveal $f$. We measured the detection ratios of our faults relative to our test pools; Table 5.1 shows the average ratios over all faults for each program. A ratio of .14 means that drawing a test at random from a test suite has a 14% chance of revealing a particular fault. Thus, the detection ratios shown reflect the average likelihood of revealing a given fault in P by randomly selecting a test from S. Of the 76 faults, 64 (84%) were detected by fewer than 25% of the tests in

the pools, 11 (14%) were detected by between 25% and 50% of the tests in the pools, and only one was detected by more than 50% of the tests in the pools. The average detection ratios for each program over its set of faulty versions ranged from .09 to .24; the overall average detection ratio was .14. Previous studies of fault-detection for imperative programs [17, 37] represent the set of faults detected by fewer than 25% of the tests (*Quartile I faults*) as "relatively difficult to detect"; the faults in our programs are predominantly Quartile I faults. This is important, because if our faults were relatively easy to detect, it could be difficult to determine whether fault detection results achieved in our study were due to our testing methodology.

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 $P$:

   (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 $P$.

To build our experimental infrastructure, and to partially automate our testing procedure, we developed the following tools, as C program or Unix scripts:

`progGen` – Read from a file listing test inputs for program P, and generate a lisp file that runs those tests on P in Forms/3.

`coverage` – Read a generated status file, and report coverage information, including how many times each du-association has been exercised, and the percentage of du-associations exercised.

`makesuite` – Select a du-validation-adequate test suite from the test pool.

`makemany` – Randomly create as many du-validation-adequate test suite as requested, and make sure no two test suites are identical.

`detector` – Compare the output of a correct version of a form-based program with the output of a faulty version of that program on the same set of tests and report any differences.

`solmatrix` – Generate a matrix which shows, for each test suite and each faulty version, how many inputs in that test suite reveal the fault in the faulty version.

### *5.2.3 Data and analysis*

Figure 5.1 displays the fault detection data obtained in our study. The figure shows, for each program, the percentage of faults detected by the du-validation-adequate test suites. Results for each program are depicted by a *box plot* – a standard statistical device for representing data sets [18]. 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 *interquartile range*). This range may be evenly partitioned by the crossbar, indicating an even distribution of the data in the interquartile range about the median. Alternatively, the data may be *skewed* to one side of the crossbar, indicating an uneven distribution. In yet other cases, only the crossbar appears, indicating that the box has length 0: at least half of the results were equivalent to the median result.

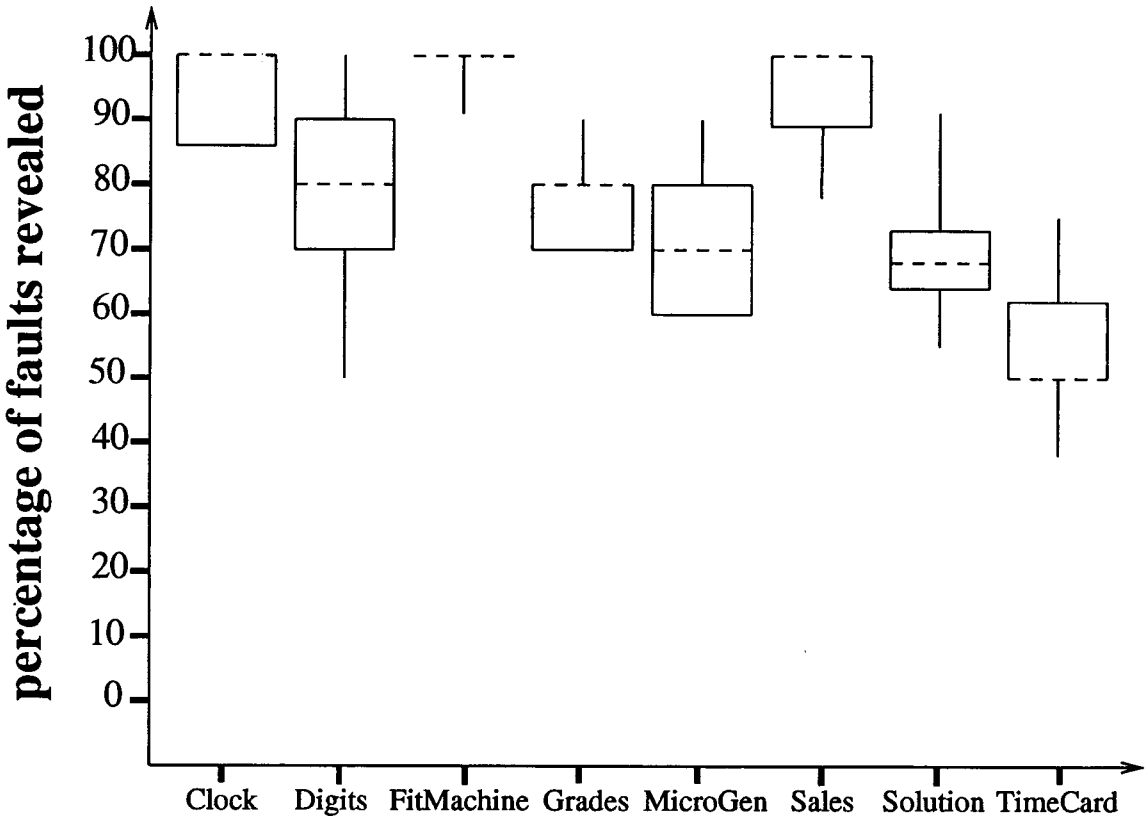


FIGURE 5.1: Faults detected by the du-validation-adequate test suites.

The whiskers that extend below and above boxes indicate ranges over which the lower 25% and upper 25% of the data, respectively, occurred. Consider, for example, the boxplot for `Digits`. This plot shows that the median test selection for `Digits` occurred at 81%, with half of the fault detection results evenly distributed between 70% and 90%, and with overall results ranging from 50% to 100%.

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 [10, 17, 25, 36, 37]. For example, in [17] the average fault detection effectiveness reported for Quartile I faults was less than 50%, and the effectiveness observed for 95% all-uses-adequate test suites in [37], where easier-to-detect faults (only 40% in Quartile I) were utilized, ranged from 51% to 92%.

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 the particular values in our test suites are representative of those that would be entered as inputs 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 had hypothesized in [31] that for form-based programs, the incidence of such associations would be lower than for imperative programs. To test this hypothesis, we counted the nonexecutable du-associations in each of our subject programs. Table 5.2 displays our results.

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 [36]. Nevertheless, the presence of these associations could be difficult to explain to users. On inspecting the nonexecutable associations in our programs, however,

| *Program* | *DUAs.* | *Non-Exe. DUAs.* | *Pct. Non-Exe.* |
|---|---|---|---|
| Clock | 64 | 9 | 15.1 |
| Digits | 89 | 28 | 31.5 |
| FitMachine | 121 | 24 | 19.8 |
| Grades | 55 | 0 | 0.0 |
| MicroGen | 31 | 3 | 9.7 |
| Sales | 28 | 0 | 0.0 |
| Solution | 32 | 2 | 6.2 |
| TimeCard | 92 | 10 | 10.9 |

TABLE 5.2: Nonexecutable du-associations.

we discovered that they could all be relatively easily determined by inspection to be nonexecutable, because of the presence of explicit contradictory conditions in defining and using cell formulas. Future work will consider whether techniques for determining (approximately) path feasibility (e.g. [7]) can operate cost-effectively behind the scenes to address this problem.

# Chapter 6

# 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 the following four features:

- Our methodology accommodates the dependence-driven evaluation model, and is compatible with evaluation engine optimizations, such as varying evaluation orders and value caching schemes.

- 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.

- 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

adding more than a constant factor is `Validate`, whose cost is the same order as the cost of recalculating the cell being validated.

- Our methodology 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. Thus, we believe our testing methodology is appropriate for use by a wide range of programmers, including the many end users who use spreadsheets.

Our empirical results suggest that in practice, our methodology can achieve fault detection results at least comparable to the results 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.

# BIBLIOGRAPHY

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

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

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

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

[5] M. Burnett and A. Ambler. Interactive visual data abstraction in a declarative visual programming language. *Journal Visual Language and Computation*, 5(1), March 1994.

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

[7] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions Software Engingeering*, 2(3), September 1976.

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

[9] E. Duesterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Proceedings 2nd Irvine Software Symposium*, March 1992.

[10] P. Frankl and S. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

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

[12] H. Gottfried and M. Burnett. Graphical definitions: Making spreadsheets visual through direct manipulation and gestures. In *IEEE Symposium Visual Language*, September 1997 (to appear).

[13] User Interface Software Group. Garnet Reference Manual, Version 2.1. Technical report, School of Computer Science at Carnegie Mellon University, November 1992.

[14] R. Gupta, M. J. Harrold, and M. L. Soffa. Program slicing-based regression testing techniques. *Journal Software Testing, Verification, and Reliability*, 6(2):83–112, June 1996.

[15] M. J. Harrold and M. L. Soffa. An incremental approach to unit testing during maintenance. In *Proceedings Conference Software Maintenance*, pages 362–367, October 1988.

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

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

[18] R. Johnson. *Elementary Statistics.* Duxbury Press, Belmont, CA, sixth edition, 1992.

[19] A. Kay. Computer software. *Scientific American*, September 1984.

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

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

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

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

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

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

[26] L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, December 1989.

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

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

[29] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings 1994 International Symposium Software Testing and Analysis*, pages 169–184, August 1994.

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

[31] G. Rothermel, L. Li, and M. Burnett. Testing strategies for form-based visual programs. In *The Eighth International Symposium Software Reliability Engineering*, November 1997 (to appear).

[32] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs. Technical Report 97-60-12, The Oregon State University, September 1997.

[33] N. Shu. *Visual Programming.* Van Nostrand Reinhold, New York, 1988.

[34] E. J. Weyuker. On testing non-testable programs. *The Comp. J.*, 15(4):465–470, 1982.

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

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

[37] W. Wong, R. Horgan, S. London, and A. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th International Conference Software Engineering*, pages 41–50, April 1995.

[38] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur. Effect of test set size and block coverage on the fault detection effecti veness. In *The Fifth International Symposium on Software Reliability Engineering* , pages 230–8, November 1994.

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

# APPENDIX

# Appendix

## MAIN SOURCE CODE FOR IMPLEMENTATION

```
;;;;  DefTable utilities

(defun make-DefTable (&key size)
  (if size (make-hash-table :size size :test 'equalp)
    (make-hash-table :test 'equalp)))

;;;;  Given Infotype(static-def or dynamic-def) and formcell
;;;;  return the found list or nil if not found

(defun DefTable-find (Infotype formName formcell)
   (let* ( (is-static-defs (eql InfoType :static-def))
           (is-dynamic-def (eql InfoType :dynamic-def))
         )
   (cond
      ((and (eql is-static-defs nil)
            (eql is-dynamic-def nil))
         (format t "illegal type information"))
      (t (setf result (gethash formcell (displayable-defTable
                                    (formtable-find formName))))
        (if result
          (cond (is-static-defs (def-header-static-defs result))
                (is-dynamic-def (def-header-dynamic-def result)))
        )
      )
   )
 )
)

;;; Add information to DefTable, replacing the old information
;;;
(defun DefTable-add (InfoType formName formcell Info)
   (let ( (is-static-defs (eql InfoType :static-def))
          (is-dynamic-def (eql InfoType :dynamic-def))
          (DefTable (displayable-defTable (formtable-find formName)))
        )
    (cond
      ((and (eql is-static-defs nil)
            (eql is-dynamic-def nil))
         (format t "illegal type information"))
      (t (setf result (gethash formcell DefTable))
           (if result
              ;;; cell has a DUheader
              (progn
                 (cond (is-static-defs
                       (setf (def-header-static-defs result) Info))
```

```
                        (is-dynamic-def
                       (setf (def-header-dynamic-def result) Info)))
                   (setf (gethash formcell DefTable) result)
               )
                ;;;else
               (setf (gethash formcell DefTable)
                  (cond (is-static-defs
                          (make-def-header :static-defs Info))
                        (is-dynamic-def
                          (make-def-header :dynamic-def Info)))) );if
          )
      ) ;;cond
  );;;let
)

;;; Append information to the old information in
;;; the DefTable

(defun DefTable-append (InfoType formName formcell Info)
   (let ( (is-static-defs (eql InfoType :static-def))
          (is-dynamic-def (eql InfoType :dynamic-def))
          (DefTable (displayable-defTable (formtable-find formName)))
        )
    (cond
      ((and (eql is-static-defs nil)
            (eql is-dynamic-def nil))
         (format t "illegal type information"))
      (t (setf result (gethash formcell DefTable))
           (if result
              ;;; formcell has a DUheader
              (cond
               (is-static-defs
                   (progn (if (def-header-static-defs result)
                              (setf (def-header-static-defs result)
                              (append (def-header-static-defs result) Info)
                              (setf (def-header-static-defs result) Info))
                              (setf (gethash formcell DefTable) result)
                         ))

                (is-dynamic-def (progn (if (def-header-dynamic-def result)
                                           (setf (def-header-dynamic-def result)
                                             (append (def-header-dynamic-def result) Info))
                                             (setf (def-header-dynamic-def result) Info))
                                              (setf (gethash formcell DefTable) result))
                                               )
               );;cond
               ;;;else
             (setf (gethash formcell DefTable)
                (cond (is-static-defs
                      (make-def-header :static-defs Info))
                      (is-dynamic-def
                      (make-def-header :dynamic-def Info)))) );if
          )
      ) ;;cond
  );;;let
)
```

```
;;; Delete the information in the DefTable given cell and InfoType.
;;;
(defun DefTable-delete (InfoType formName formcell)
   (let* ( (is-static-defs (eql InfoType :static-def))
           (is-dynamic-def (eql InfoType :dynamic-def))
           (DefTable (displayable-defTable (formtable-find formName)))
         )

     (setf  result (gethash formcell DefTable))
     (if result
         (progn
              (cond (is-static-defs (setf (def-header-static-defs result)
                                          nil ))
                    (is-dynamic-def (setf (def-header-dynamic-def result)
                                          nil ))
                    (setf (gethash formcell DefTable) result)
               )
          )
          nil
     );;If
    ) ;;Let
)

;;;; Clear all entry in DefTable
(defun DefTable-clear (formName)
  (setf DefTable (displayable-defTable (formtable-find formName)))
  (clrhash DefTable)
  DefTable
)

;;; Displays all the entries in a form's DefTable, used for debugging.
;;;

(defun DefTable-list (formName)

  (setf DefTable (displayable-defTable (formtable-find formName)))
  (format t "~%Total number of hash table entries = ~a~%"
            (hash-table-count DefTable))

  (maphash #'(lambda (key val)
      (progn
        (format t "~%~a : ~a static-defs and ~a dynamic-def~%"
           key
           (length (def-header-static-defs val))
           (length (def-header-dynamic-def val)))
           (format t "  static-defs :~%")
           (if (eql 0 (length (def-header-static-defs val)))
           (format t "    NONE~%")
           (dolist (sdef (def-header-static-defs val))
             (format t "    ~a~%" sdef)))
           (format t "  Dynamic-def :~%")
           (if (eql 0 (length (def-header-dynamic-def val)))
           (format t "    NONE~%")
           (dolist (ddef (def-header-dynamic-def val))
             (format t "    ~a~%" ddef)))))
    DefTable)
)
```

```
;;; Given a fom, find out all its cells' static
;;;; definitions and save them
;;;  into the DefTable

(defun DefTable-add-static-defs (form)

  (maphash #'(lambda (cellID value)
           (get-all-static-defs cellID form)
      );;lambda
      (displayable-celltable (formtable-find form)))

      (DefTable-AddSingleStaticToDynamic form)
);;end  defun

;;;; DUpairTable utilities
;;;;

(defun make-DUpairTable (&key size)
  (if size (make-hash-table :size size :test #'equalp)
    (make-hash-table :test #'equalp)))

;;;; Add a du pair to a form's DUpair table
(defun DUpairTable-add (DUpair formName)
  (setf DUpairTable (displayable-DUpairTable
              (formtable-find formName)))
  (setf result (gethash DUpair DUpairTable))
  (if result
      nil
      (setf (gethash DUpair DUpairTable)
            (make-du-header :status nil)
      )
  )
)

;;;; set DUpair status to be T. used in validation-backtracking
;;;;
(defun DUpairTable-mark (DUpair formName)
  (setf DUpairTable (displayable-DUpairTable
                (formtable-find formName)))
  (setf result (gethash DUpair DUpairTable))
  (if result
      (progn
      (setf (du-header-status result) t)
      (setf (gethash DUpair DUpairTable) result)
      );progn
  );if
)

;;;; set all du pairs in DUpairTable to be nil status.
;;;;
(defun DUpairTable-unmarkall (formName)
       (setf DUpairTable (displayable-DUpairTable
                     (formtable-find formName)))
       (maphash #'(lambda (key val)
          (setf (du-header-status val) nil)
        )
        DUpairTable)
```

```
)

;;;; clear all entries in DUpairTable
;;;;
(defun DUpairTable-clear (formName)
  (setf DUpairTable (displayable-DUpairTable
                (formtable-find formName)))
  (clrhash DUpairTable)
  DUpairTable
)

;;;; Display all marked du pairs in DUpairTable
;;;;
(defun DUpairTable-list-marked (formName)
  (setf DUpairTable (displayable-DUpairTable
                (formtable-find formName)))
    (format t "Marked DU pairs: ~%")
    (maphash #'(lambda (key val)
      (if (du-header-status val)
       (format t "~% ~a ~%" key)
       ))
     DUpairTable)
)

;;;; Display all unmarked du pairs in DUpairTable
;;;;
(defun DUpairTable-list-unmarked (formName)
    (setf DUpairTable (displayable-DUpairTable
                (formtable-find formName)))
    (format t "Unmarked DU pairs: ~%")
        (maphash #'(lambda (key val)
        (if (du-header-status val)
         nil
        (format t "~% ~a ~%" key)
    ))
    DUpairTable)
)

;;;; Display all du pairs in DUpairTable
;;;;
(defun DUpairTable-list (formName)
    (setf DUpairTable (displayable-DUpairTable
                (formtable-find formName)))
    (format t "~%Total number of hash table entries = ~a~%"
    (hash-table-count
     DUpairTable))

    (format t " DUpair:      Status:  ~%")

    (maphash #'(lambda (key val)
       (format t "~%~a : ~a  ~%" key
         (du-header-status val)
        ))
        DUpairTable)
)

;;;; A list of two elements is return.
```

```
;;;; The first element shows the total
;;;; number of DUpairs that the given formcell
;;;; is the use.
;;;; The second element shows the number
;;;; of marked DUpairs that the given formcell
;;;; is the use

(defun DUPairTable-cellsIn (formName formcell)
      (let ( (inTotal 0)
             (inMarked 0)
             (DUpairTable (displayable-DUpairTable
                          (formtable-find formName)))
             )

             (maphash #'(lambda (key val)
              (if (equalp formcell (third key))
                  (progn
                        (setf inTotal (+ inTotal 1))
                        (if (du-header-status val)
                            (setf inMarked (+ inMarked 1))
                            )
                            )
                          ))
                          DUpairTable)
          (list inTotal inMarked)
   )
)

;;;; Given a form, build its DUpairTable.
;;;; Main function to add all the form's DU pairs into DUpairTable
(defun DU-add-pairs (form)

(maphash #'(lambda (cellID value)
        (if cellID
           (get-all-dupairs cellID form)
           nil
         )) ;;lambda
   (displayable-celltable (formtable-find form)))
)


;;;;  given a cell, get all its static definitions
(defun get-all-static-defs (cellID form)
      (let* (
             (alist ;; a list of nested if-then-else structure
                 (parse-if-then-else
                  (read-from-string
                  (parse-prepare-string
                  (unparse-exp ;;; unparse parsed string
                  (displayable-generalFormula
                  (displayable-getCell form cellID))
                  (formtable-find form)))
                 ))))

      (setf formcell (format nil "~a%~a" form cellID))
      (setf aStack nil)
      (if (listp alist)
          (get-defs alist formcell)
```

```
      )
    );;let*
)

;;;; given a cell, calculate all its DU pair
;;;; and save them into DUpairTable.
(defun get-all-dupairs (cellID form)
       (let* ((cellEntry nil)
              (alist (parse-if-then-else
                     (read-from-string
                     (parse-prepare-string
                     (unparse-exp
                     (displayable-generalFormula
                     (displayable-getCell form cellID))
                     (formtable-find form)))
                     )))
             )
             (setf cellEntry (format nil "~a%~a" form cellID))
             (setf aStack nil)
             (if (listp alist)
                 (get-dus alist cellEntry)
             )
    )
)


;;;; Calculate a cell's static defintions and save
;;;; into DefTable whenever a new one is found
;;;; a defintion is a path from the root to an end node.
;;;; aStack is used to keep path
;;;; alist is a nested if-then-else structure list
(defun get-defs (alist formcell)
     (let* (
             (condition (car alist))
             (Tpart (second alist))
             (Fpart (third alist))
             (formName (parse-isolate-formName formcell))
           )

           (cond
                ( (eq condition nil)
                        (progn
                            (if (listp Tpart)
                            (push Tpart aStack)
                            (push (list Tpart) aStack))
                  ;; save into table when a new path is found
                  (DefTable-append :static-def formName
                             formcell (list (reverse aStack)))
                                    (pop aStack)
                                );;progn
                 ) ;;nil condition
                 (t (progn
                        (push condition aStack)
                        (push 'T aStack)   ;;; Take true path
                        (get-defs Tpart formcell) ;; call recursively
                        (pop aStack)
```

```
                        (push 'F aStack)    ;;; Take false path
                        (get-defs Fpart formcell);; call recursively
                        (pop aStack)
                        (pop aStack)
                        );;progn
                        );;t
          ) ;;end cond
     );; end let*
)

;;; given a cell's formula list, find out all the cells referenced there
;;; A list of referenced form%cellID  is returned

(defun get-uses (alist)
   (let* ( (formpart nil)
           (cellpart nil)
           (cellID  nil)
           (formcellID nil)
         )
   (if (listp alist)
         (dolist (asub alist)
               (progn
                  (if (listp asub)
                      (get-uses asub)
                      (if (symbolp asub)
                         (progn
                          (setf formpart
                                 (parse-isolate-formName (string asub)))
                          (if formpart
                              ;; is cell name.
                             (progn
                                (setf cellpart
                                (subseq (string asub) (+ (length formpart) 1)))
                                 (if (gethash (read-from-string cellpart)
                                     (displayable-cellTable
                                               (formtable-find formpart)))
                                     ;; cellpart is cellID
                                     (setf cellID cellpart)
                                     ;; cellpart is cell Name, get cellID
                                     (setf cellID (car
                                           (gethash cellpart
                                            (displayable-nameTable
                                           (formtable-find formpart))
                                    ))));;endif
                                 (setf formcellID
                                       (format nil "~a%~a" formpart cellID))
                                 (setf uses (append uses (list formcellID)))
                               );;end progn
                             );;end if
                           );;end progn
                          );;end if
                          );;end if
                     );;end progn
         ) ;;dolist
         ;;else
         (if (symbolp alist)
```

```
            (progn
                (setf formpart (parse-isolate-formName (string alist)))
                (if formpart
                     ;; is a cell
                    (progn
                        (setf cellpart
                         (subseq (string alist) (+ (length formpart) 1)))
                        (if (gethash cellpart
                         ;; cellpart is cellID
                             (displayable-cellTable
                                  (formtable-find formpart)))
                                (setf cellID cellpart)
                          ;; cellpart is cell Name, get cellID
                        (setf cellID (car
                        (gethash cellpart
                        (displayable-nameTable
                        (formtable-find formpart))))));;end if
                       (setf formcellID (format nil "~a%~a" formpart cellID))
                       (setf uses (append uses (list formcellID)))
                            );;end progn
                       );;end if
           );;progn
        ) ;;if
      );;if
      uses
   )
)

;;; return its static-defintion if it's a cell
;;; otherwise return nil
(defun get-cell-definitions (name)
    (setf formName (parse-isolate-formName name))
    (if (stringp name)
        (DefTable-find :static-def formName name)
         (if (symbolp name)
             (DefTable-find :static-def formName (symbol-name name)))
     )
)

;;; calculate a cell's all DUpairs and
;;; save them into DUpairTable
(defun get-dus (alist formcell)
       (let* (
                  (condition (car alist))
                  (Tpart (second alist))
                  (Fpart (third alist))
                  (formName (parse-isolate-formName formcell))
             )
             (cond
                ((eq condition nil)
                 (if (not (equalp Tpart nil))
                     (progn
                       (if (not (listp Tpart))
                           (push (list Tpart) aStack)
                                 ;; to match single cell Ref
                            (push Tpart aStack)
```

```
                    ) ;;if
                    (setf uses nil)
                    (dolist (cellUsed (get-uses Tpart)) ;;; get uses
                       (progn
                        (dolist
                         (defs (get-cell-definitions cellUsed)) ;;; get defs
                        (DUpairTable-add
                           (list cellUsed defs (read-from-string formcell)
                             (reverse aStack)) formName)) ;;; pair Def-use

                         );progn
                        )
                        (pop aStack)
                        );progn
                        );if
                        ); nil condition
               (t (progn
                  (push condition aStack)
                  (push 'T aStack)        ;;;; take True path
                  (setf uses nil)
                  (dolist (cellUsed (get-uses condition)) ;;; get uses
                  (progn
                   (dolist
                     (defs (get-cell-definitions cellUsed)) ;;; get defs
                     (DUpairTable-add
                         (list cellUsed defs (read-from-string formcell)
                         (reverse aStack)) formName))  ;;; pair def-use
                                    );;progn
                                    );;dolist
                  (get-dus Tpart formcell)
                  (pop aStack)
                  (push 'F aStack)      ;;; take False path
                  (setf uses nil)
                  (dolist (cellUsed (get-uses condition))  ;;; get uses
                  (progn
                   (dolist
                      (defs (get-cell-definitions cellUsed)) ;;; get defs
                      (DUpairTable-add
                      (list cellUsed defs (read-from-string formcell)
                      (reverse aStack)) formName))    ;;; pair def-use
                                    );;progn
                                    );;dolist
                  (get-dus Fpart formcell)
                  (pop aStack)
                  (pop aStack)
                  ); progn
                  ); t

            );cond
      );let
);defun


;;;;  Take a list of expression from a cell's formula string,
;;;;  Convert it to a if-then-else list, which is a three
;;;;  elements list, the first element is IF-part, which is
```

```
;;;;  a condition expression, the second is THEN-part, the third
;;;;  is ELSE-part, THEN-part and ELSE-part can be expression
;;;;  or if-then-else list.

(defun parse-if-then-else (anExp)
  (let ( (tempLength (if (listp anExp) (length anExp) 1))
       )
    (cond
         ;;empty expression
           ((null anExp)
            (list nil 'I nil))
         ;;  singleton
        ((atom anExp)
           (list nil anExp nil))

        ( (and (listp anExp) (not (eq (car anExp) 'if)))
             (list nil anExp nil))
         ;;;if-then-else
         (
              (and (eq (car anExp) 'if)
              (eq (third anExp) 'then))
                (cond
                  ((= tempLength 4) ;;; else-less
                   (list (second anExp)
                          (parse-if-then-else (fourth anExp))
                   )
                  )
                 ((and (= tempLength 6) (eq (fifth anExp) 'else))
                   ;;; if-then-else complete
                 (list (second anExp)
                         (parse-if-then-else (fourth anExp))
                         (parse-if-then-else (sixth anExp)))
                 )
                 (t  (list 'Error))
               )
             )
         )
    )
)
```

Source code is inserted in some parts of Forms/3's evaluation engine, mainly in formsIf and Same as functions.
Now they are like following:

```
(defun formsIf (arg-list form cell seeTime &key debug)
  (let* ((ans $aNoValueDycon) (defTime seeTime)
         (expireTime (time-addOne seeTime)) (termTime nil)
         (Aref (car arg-list))
         (Aans nil) (AdefTime nil) (AexpireTime nil) (AtermTime nil)
         (Bref (cadr arg-list))
         (Bans nil) (BdefTime nil) (BexpireTime nil) (BtermTime nil)
         (Cref (caddr arg-list))
         (Cans nil) (CdefTime nil) (CexpireTime nil) (CtermTime nil)
         (cellID  (displayable-id cell)) (cellEntry nil)
         (formName (displayable-name form))
         )
```

```
(setf cellEntry (format nil "~a%~a" (displayable-name form) cellID))

(multiple-value-setq (Aans AdefTime AexpireTime AtermTime)
                     (arg-process Aref cell form seeTime :debug debug))
(cond ((or (null Aans) (typep Aans 'noValueDycon))
       (multiple-value-setq (ans defTime expireTime termTime)
                  (values $aNoValueDycon AdefTime AexpireTime AtermTime)))

      ;; condition is true -- return the 'then' arg
      ((equalp (intrinsic-value Aans) t)
           (DefTable-append :dynamic-def formName cellEntry
              (list (read-from-string (parse-prepare-string
                           (unparse-exp Aref form))) 'T))
         (cond ((listp Bref)
                (if (equalp (car Bref) 'if)
                nil
                (DefTable-append :dynamic-def formName cellEntry
                          (list (read-from-string (parse-prepare-string
                                (unparse-exp Bref form))))))
                )

            (t
              (DefTable-append :dynamic-def formName cellEntry
                        (list (read-from-string (parse-prepare-string
                                (unparse-exp Bref form)))))
          )) ;;end cond

       (multiple-value-setq (Bans BdefTime BexpireTime BtermTime)
                    (arg-process Bref cell form seeTime :debug debug))
       (multiple-value-setq (ans defTime expireTime termTime)
                           (values Bans
                                   (timeMax AdefTime BdefTime)
                                   (timeMin AexpireTime BexpireTime)
                                   (if (and AtermTime BtermTime
                                            (time= AtermTime AdefTime))
                                       (timeMax AtermTIme BtermTime)
                                     nil)))
       )

      ;; condition is false -- return the 'else' arg (if it exists)
      ((equalp (intrinsic-value Aans) nil)
       (if Cref
         (progn
             (DefTable-append :dynamic-def formName cellEntry
                (list (read-from-string (parse-prepare-string
                                (unparse-exp Aref form))) 'F))

           (cond
               ((listp Cref)
                (if (equalp (car Cref) 'if)
                    nil
                    (DefTable-append :dynamic-def formName cellEntry
                        (list (read-from-string (parse-prepare-string
                                (unparse-exp Cref form)))))
                 )) ;; end listp
              (t
```

```
                    (DefTable-append :dynamic-def formName cellEntry
                                (list (read-from-string (parse-prepare-string
                                        (unparse-exp Cref form)))))
                          );;end  t
                    ) ;; end cond

                (multiple-value-setq (Cans CdefTime CexpireTime CtermTime)
                          (arg-process Cref cell form seeTime :debug debug))
                         (multiple-value-setq (ans defTime expireTime termTime)
                                  (values Cans
                                          (timeMax AdefTime CdefTime)
                                          (timeMin AexpireTime CexpireTime)
                                          (if (and AtermTime CtermTime
                                                   (time= AtermTime AdefTime))
                                              (timeMax AtermTime CtermTime)
                                            nil)))) ;end progn, if
                          (DefTable-append :dynamic-def formName cellEntry
                          (list (read-from-string (parse-prepare-string
                           (unparse-exp Aref form))) 'F))
                );; if
        )

         ;; condition is neither true, false, nor missing -- error
         ( t
           (cond
            ((typep Aans 'errorDycon)
             (multiple-value-setq (ans defTime expireTime termTime)
               (values Aans AdefTime AexpireTime AtermTime)))
            (t
             (multiple-value-setq (ans defTime expireTime termTime)
               (values $aTypeErrorDycon AdefTime AexpireTime AtermTime)))))
         ) ;end cond

    (values ans defTime expireTime termTime)
)) ;end let, defun FormsIf


(defun <- (listofx form cell seeTime &key debug)
  (let* ((x (car listofx))
         (cellID (displayable-id cell))
         (cellEntry nil)
         (formName (displayable-name form))
         )

         (progn
           (if (not (equalp formName "System"))
             (progn
              (setf cellEntry (format nil "~a%~a" formName cellID))
               (DefTable-append :dynamic-def
                formName cellEntry (list (list x)))
             );;progn
           );;if
          (arg-process x cell form seeTime :debug debug)
       );;progn
))
```

```
;;;; Main function to do validation backtracking
;;;; set up the environment and invoke validation
;;;; algorithm
(defun vb (formName cellID)

     (DefTable-AddSingleStaticToDynamic formName)
        ;; check if dynamic-def all available.
       (DUpairTable-unmarkall formName)
        ;; clear all the marked status
       (if (equalp $CurrentVisitedNumber $MAXINT)
                  (setf $CurrentVisitedNumber 1)
                  (setf $CurrentVisitedNumber
                    (+ 1 $CurrentVisitedNumber))
                  );end if
       (setf formcell (format nil "~a%~a" formName cellID))
       (valid-back formcell)
)

;;;; Validate formcell
;;;; Backtrack from this cell recursively
;;;; A list of marked DU pairs is returned.
(defun valid-back (formcell)
    (let* (
      (formName (parse-isolate-formName formcell))
      (currentDef (DefTable-find :dynamic-def formName formcell))
      (path nil)
        )
   (setf validList nil)
   (setf path nil)
   (setVisited formcell)
   (dolist (oneNode currentDef)
       (if (listp oneNode)
       (progn
       (setf pred (second (member oneNode currentDef)))
       (setf path (append path (list oneNode)))
       (if pred (setf path (append path (list pred))))
          (setf uses nil)
          (dolist (cellUsed (get-uses oneNode))
          (progn
          (setf formName (parse-isolate-formName cellUsed))
          (setf def (DefTable-find :dynamic-def formName cellUsed))
               (if (not (member
               (list cellUsed def (read-from-string formcell) path)
                        validList))
                (setf validList
                 (append validList
                 (list (list (read-from-string cellUsed) def ;; definition
                  (read-from-string formcell) path))))) ;; use

                 (DUpairTable-mark
                 (list cellUsed def
                 (read-from-string formcell) path) formName);;DupairMark
                 (setf formName (parse-isolate-formName cellUsed))
                 (setf cellID (subseq cellUsed (+ (length formName) 1)))
                 (if (<
                   (displayable-visitedNum
                   (displayable-getcell formName cellID))
```

```
                    $CurrentVisitedNumber
                   )
                   (setf validList
                   (append validList
                   (valid-back cellUsed)))
                )
             );progn
            );dolist
           );progn
          );;end-if
         );end-dolist
       validList  ;; return
   );end-let*
)

;;;; mark this formcell as visited to avoid revisiting
;;;; used in validation backtracking process
(defun setVisited (formcell)
       (setf formName (parse-isolate-formName formcell))
       (setf cellID (parse-isolate-cellname formcell))
       (setf (displayable-visitedNum
             (displayable-getCell formName cellID))
              $CurrentVisitedNumber)
)
```