An Abstract of the Thesis of

Bing Ren for the degree of Master Science in Computer Science presented on June 11, 2001.
Title: Extensions to the WYSIWYT Methodology

Abstract approved: Redacted for privacy

Margaret M. Burnett

Researchers in the Forms/3 group have previously developed the WYSIWYT methodology, exploring a way of systematically testing spreadsheet languages. The previous work presented the WYSIWYT methodology for individual spreadsheet cells, and later partially extended it to large grids in which some cells share the same formula. The Forms/3 spreadsheet language supports not only operations of individual cells but also some advanced programming features such as homogeneous grids, recursive programs, and user-defined abstract data types. Therefore, it is important for the testing methodology to support even these more powerful features of the language, not just the "easy parts". In this document, we present extensions to the WYSIWYT methodology for these advanced features. We optimized the visual aspects of testing spreadsheet grids and collected experimental information about scalability. We also developed two possible ways the WYSIWYT methodology could be extended to accommodate recursion in terms of their testing theoretic aspects, implementation strategies, algorithms and time complexities. Since the ultimate goal is to help the people using these languages, we also conducted an empirical study and used its results to inform our choice as to which of these two approaches to adopt. Finally, we developed an approach of testing user-defined abstract data types; here we present design, implementation issues, algorithms and time complexities.

Extensions to the WYSIWYT Methodology

by

Bing Ren

A Thesis Submitted
to
Oregon State University

In partial fulfillment of
the requirements for the
degree of

Master of Science

Completed June 11, 2001
Commencement June 2002

Master of Science thesis of Bing Ren presented on June 11, 2001.

APPROVED:

Redacted for privacy

Major Professor, representing Computer Science

Redacted for privacy

Head of Department of Computer Science

Redacted for privacy

Dean of 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

Bing Ren, Author

Acknowledgment

I am grateful to my major professor, Dr. Burnett of the important ideas, useful hints, precious comments she puts in my work. This thesis was extensively discussed in a series of meetings with her. My appreciation goes to Dr. Rothemel, Dr. Cook and Dr. Wood for their helps and being my committee. Thanks also go to all Forms/3 group members who gave me their valuable suggestions.

Table of Contents

Table of Contents (Continued)

List of Figures

List of Figures (Continued)

List of Figures (Continued)

List of Tables

List of Tables (Continued)

Extensions to the WYSIWYT Methodology

Chapter 1: Introduction

Testing is an important activity, used widely by professional and end-user programmers alike in locating faults in their programs. In recognition of its importance and widespread use, there has been extensive research into effective testing in traditional programming languages in the imperative paradigm (e.g., [Duesterwald et al. 1992, Frankl and Weyuker 1998, Harrold and Soffa 1988, Laski and Korel 1993, Ntafos 1984, Offutt et al. 1996, Perry and Kaiser 1990, Rapps and Weyuker 1985, Rothermel and Harrold 1997, Wang and Ambler 1996, Weyuker 1993]). However, there are few reports in the literature on testing in other paradigms, and no reports (with the exception of our own previous work) that we have been able to locate on testing in spreadsheet systems. The spreadsheet paradigm includes not only commercial spreadsheet systems, but also a number of research languages that extend the paradigm with features such as gestural formula specification [Burnett and Gottfried 1998, Leopold and Ambler 1997], graphical types [Burnett and Gottfried 1998, Wilde and Lewis 1990], visual matrix manipulation [Wang and Ambler 1996], high-quality visualizations of complex data [Chi et al. 1998], and specifying GUIs [Myers 1991]. In this document, we use the term *spreadsheet languages* to describe all such systems following the spreadsheet paradigm and the term *extended spreadsheet languages* to describe spreadsheet languages, such as those in the previous sentence, with features beyond those of widely-used commercial spreadsheet systems.

Despite the perceived simplicity of the spreadsheet paradigm, research shows that many spreadsheets contain faults. A recent survey of the relevant literature [Panko and Halverson 1996] provides details: for example, in four field audits of operational spreadsheets, errors were found in an average of 20.6% of spreadsheets audited; in eleven experiments in which participants created spreadsheets, errors were found in an average of 60.8% of the spreadsheets; and in four experiments in which the participants inspected spreadsheets for errors, the participants missed an average of 55.8% of the errors.

To help solve this problem, in previous work [Rothermel et al. 2001, Rothermel et al. 1997, Rothermel et al. 1998, Sheretov 2000], we presented a testing methodology for spreadsheets termed the "What You See Is What You Test" (WYSIWYT) methodology. The WYSIWYT methodology provides feedback as to the "testedness" of cells in simple non-recursive spreadsheets in a manner that is incremental, responsive, and entirely visual. However, scalability to large grids was only partially successful, and testing on recursive spreadsheets as well as supporting user-defined abstract data types were not addressed in that previous work. In this document, we provide extensions to allow the WYSIWYT methodology to address these deficiencies.

Chapter 2: Background

2.1 Spreadsheet languages

Perhaps the most widely used programming paradigm today is the spreadsheet paradigm. Spreadsheet languages differ from most other commonly used programming languages in that they provide a declarative approach to programming, characterized by a dependence-driven, direct-manipulation, immediate-feedback working model [Ambler et al. 1992]. Users of spreadsheet languages "program" by specifying the contents of a spreadsheet. The contents of a spreadsheet are a collection of cells; each cell's value is defined by that cell's formula. These formulas reference values contained in other cells and use them in calculations. As soon as a cell's formula is defined, the underlying evaluation engine automatically calculates the cell's value and those of affected cells (at least those that are visible), and immediately displays new results.

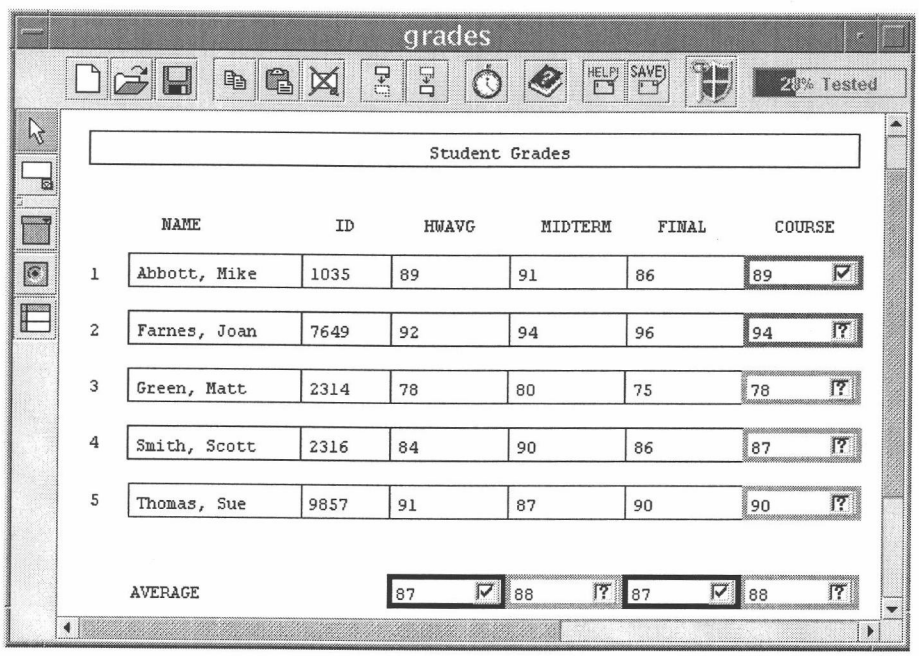


Figure 1. Spreadsheet for calculating student grades.

Spreadsheet languages are used for computational tasks ranging from simple "scratchpad" applications developed by single non-programmer users to large-scale, complex systems developed by multiple sophisticated users [Panko 1998].

In this document, we present examples of spreadsheets in the research language Forms/3 [Burnett et al. 2001]. Forms/3 is the extended spreadsheet language in which we prototyped our method. Figure 1 shows a traditional-style spreadsheet used to calculate student grades in Forms/3. The spreadsheet lists several students, and several assignments performed by those students. The last row in the spreadsheet calculates average scores for each assignment, the rightmost column calculates weighted averages for each student, and the lower-right cells gives the overall course average (formulas not shown).

In this document, we consider an advanced spreadsheet language model that includes not only ordinary spreadsheet-like formulas, but also some advanced programmer-oriented features such as large grids with shared formulas, user-defined abstract data types, and recursion. The grammar shown in Table 1 reflects a subset of Forms/3, a Turing-complete spreadsheet language following this model [Burnett et al. 2001]. The figures presented in this document were programmed using this subset.

formula	::=	*Blank* | expr
expr	::=	*Constant* | ref | infixExpr
		| prefixExpr | ifExpr | composeExpr | (expr)
infixExpr	::=	subExpr infixOperator subExpr
prefixExpr	::=	unaryPrefixOperator subExpr
		| binaryPrefixOperator subExpr subExpr
ifExpr	::=	IF subExpr THEN subExpr ELSE subExpr
		| IF subExpr THEN subExpr
composeExpr	::=	COMPOSE subExpr AT (subexpr subexpr) composeWithClause
		| COMPOSE subExpr AT (subexpr subexpr)
composeWithClause	::=	WITH subexpr AT (subexpr subexpr) composeWithClause
		| WITH subexpr AT (subexpr subexpr)
subExpr	::=	*Constant* | ref | (expr)
infixOperator	::=	+ | - | * | / | MODE | AND | OR | = | > | < | ...
unaryPrefixOperator	::=	- | ROUND | ABS | WIDTH | HEIGHT | ERROR? | ...
binaryPrefixOperator	::=	APPEND | MATRIXSEARCHROWWHERE | ...

ref	::=	cellRef | *Form.ID* : cellRef
cellRef	::=	*SimpleCell.ID* | *Matrix.ID* | *Matrix.ID* [subscripts]
		| *Abs.ID* | *Abs.ID* [*SimpleCell.ID*] | *Abs.ID* [*Matrix.ID*] | *Abs.ID*
		| [*Matrix.ID*] [subscripts]
subscripts	::=	matrixSubscript@matrixSubscript
matrixSubscript	::=	expr

Table 1. The grammar for Forms/3 formulas. Note that subexpressions are fully parenthesized, thereby avoiding ambiguity. As the top section shows, Forms/3 has the usual spreadsheet formula operators and also some operators supporting computations on grids (dynamic matrices), on graphics, on recursion and on user-defined abstract data types. The bottom section shows cell reference syntax, which includes row/column referencing for cells that are in a grid (*Matrix*).

2.2 The WYSIWYT methodology

The underlying assumption in our work has been that, as the user develops a spreadsheet incrementally, he or she could also be testing incrementally. We have integrated a prototype implementation of the WYSIWYT methodology into Forms/3 [Rothermel et al. 2001], and the examples in this document are presented in that language. In our prototype, every cell in the spreadsheet is considered to be untested when it is first created, except *input cells* (cells whose formulas may contain constants and operators, but no cell references and no `if`-expressions), which are considered trivially tested. For the non-input cells, "testedness" is reflected via border colors on a continuum from untested (red) to tested (blue).

The process is as follows. During the user's spreadsheet development, whenever the user notices a correct value, he or she lets the system know of this decision by *validating* the correct cell (clicking in the decision checkbox in its right corner), which causes a checkmark to appear, as shown in Figure 1. This communication lets the system track judgments of correctness, propagate the implications of these judgments to cells that contributed to the computation of the validated cell's value, and reflect this increase in "testedness" by coloring borders of the checked cell and its contributing cells more "tested" (more blue). On the other hand, whenever the user notices an incorrect value, rather than checking it off, he or she eventually finds the faulty formula and fixes it. This formula edit means that affected cells will now have to be re-tested; the system is aware of which ones those are, and re-colors their borders more "untested" (more red).

2.3 The CRG model, DU-Adequacy and four testing tasks

Prior to this work, researchers in our group developed an abstract testing model for spreadsheets [Rothermel et al. 1997]. This model is called a *cell relation graph* (CRG). A CRG is a pair (V, E), where V is a set of formula graphs, and E is a set of directed edges modeling dataflow relationships between pairs of elements in V. A formula graph models flow of control within a single cell's formula, and is comparable to a control flow graph. In non-recursive simple spreadsheets, there is one formula graph for each cell. The process of translating an abstract syntax tree representation of an expression into its control flow graph representation is well known [Aho et al. 1986]; a similar translation applied to the abstract

syntax tree for each formula in a spreadsheet yields that formula's formula graph. For example, Figure 2 shows a portion of the CRG for the cells in Figure 1, delimited by dotted rectangles. In these graphs, nodes labeled "E" and "X" are *entry* and *exit* nodes, respectively, and represent initiation and termination of evaluation of formulas. Nodes with multiple out-edges (represented as rectangles) are *predicate* nodes. Other nodes are *computation* nodes. Edges within formula graphs represent flow of control between expressions, and edge labels indicate the value to which condition expressions must evaluate for particular branches to be taken.

We used the cell relation graph model to define several test adequacy criteria [Rothermel et al. 1997]. The strongest criterion we defined, *du-adequacy*, is the criterion we use in this document to define when a spreadsheet has been tested "enough". The *du-adequacy criterion* is type of dataflow adequacy criterion. Such criteria relate test adequacy to

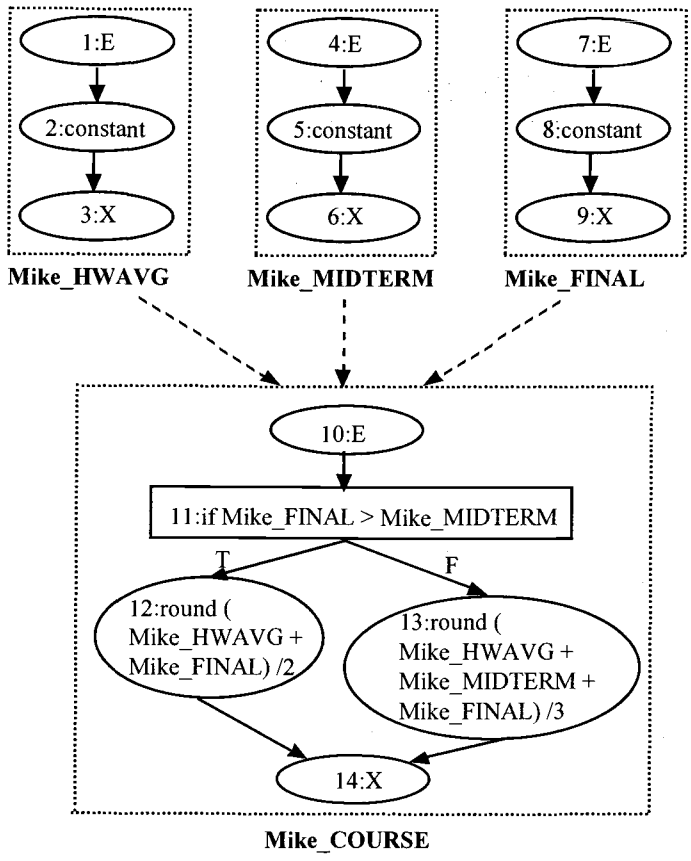


Figure 2. A partial cell relation graph of Figure 1. These are the formula graphs for Abbott, Mike.

interactions between definitions and uses of variables in source code (*definition-use associations*, abbreviated *du-associations*). In spreadsheets, cells play the role of variables; a *definition* of cell C is a node in the formula graph for C representing an expression that defines C, and a *use* of cell C is either a computational use (a non-predicate node that refers to C) or a predicate use (an out-edge from a predicate node that refers to C). Under this criterion, a cell X will be said to have been tested enough when all of its *definition-use associations* have been *covered* (executed) by at least one test. In this model, a *test* is a user decision as to whether a particular cell contains the correct value, given the inputs upon which it depends. Decisions are communicated to the system when the user checks off a cell to validate it. Thus, given a cell X that references Y, *du-adequacy* is achieved with respect to the interactions between X and Y when each of X's *uses* (references to Y) of each *definition* in Y has been covered by a test. We assume that cells whose formulas are simply constant values do not need testing, and refer to them as "input cells" in this document.

It is not always possible to test all du-associations. For example, one of Y's definitions might depend on some cell Z being less than 0, with X's use of Y occurring only if Z is greater than 0, and thus X-Y du-associations will never execute. Such du-associations are said to be *infeasible*. It is well known that infeasible elements such as these represent a problem for testing methodologies, and the WYSIWYT methodology does not include a comprehensive solution, but one design goal is to avoid exacerbating the problem in the methodology.

To support the WYSIWYT testing methodology, the system needs to keep the information described in Table 2 for each cell. With this information, the system performs four tasks, each of which is triggered by a user action:

- **Task 1:** When the user edits a formula of a cell C, static du-associations are collected for the cell, the collection of which is denoted C.DUA.
- **Task 2:** When a cell C is executed, the most recent execution trace of its nodes, denoted C.Trace, is stored (via a probe in the evaluation engine).
- **Task 3:** When the user validates C by checking it off, C.Trace is consulted to find which of the du-associations in C.DUA should be marked "covered".
- **Task 4:** When the user edits a formula for a (non-input) producer of C, C.DUA's du-associations need to be reset to "not covered".

	Information collected	Description	Collected statically or dynamically
1	*C.DirectProducers*	The cells referenced explicitly in *C*'s formula.	Statically
2	*C.DirectConsumers*	The cells whose formulas explicitly reference *C*.	Statically
3	*C.Defs*	The definitions explicitly present in *C*'s formula.	Statically
4	*C.Uses*	The uses explicitly present in *C*'s formula.	Statically
5	*C.DUAs:*	A set of du-associations, consisting of:	
	C.DUAs.Incoming	All du-associations whose uses are in *C.Uses*.	
	C.DUAs.Outgoing	All du-associations whose definitions are in *C.Defs*.	
	C.DUA:	An element of *C.DUAs*, in format (definition, use, exercised), consisting of:	
	C.DUA.definition	The definition.	Statically
	C.DUA.use	The use.	Statically
	C.DUA.exercised	True if *C.DUA* has been exercised; otherwise false.	Dynamically
6	*C.Trace*	The set of *C*'s formula graph nodes that were executed in the most recent evaluation of *C*.	Dynamically

Table 2. The primary testing information collected for each cell *C*. It is collected by updating hash tables while parsing formulas (statically) and while executing formulas (dynamically).

2.4 Testing homogeneous spreadsheet grids

The methodology for testing spreadsheets as described above worked at the granularity of individual cells. However, most large grids in spreadsheets are fairly homogeneous, i.e., they consist of many cells whose formulas are identical except for some of the row/column indices. In this document, the term *grid* implies some homogeneity, and the term *region* refers to a subgrid in which every cell has the same formula, except that row/column indices may differ.

Our work was prototyped using a grid called a matrix in Forms/3. To define values for a Forms/3 grid's (matrix's) cells, the user statically partitions the grid into rectangular regions and, for each region, enters a single formula for all cells in it. To statically derive a cell's formula from its shared region formula, any "pseudo-constants" *i* and *j* in the formula are replaced by the cell's actual row and column number. Each grid has two additional cells, its row dimension cell and column dimension cell to specify its number of rows and columns. These cells can have arbitrarily complex formulas. Figure 3 shows a spreadsheet similar to that in Figure 1 rewritten with the use of grids. The row and column dimension formulas are simply constants in this example.

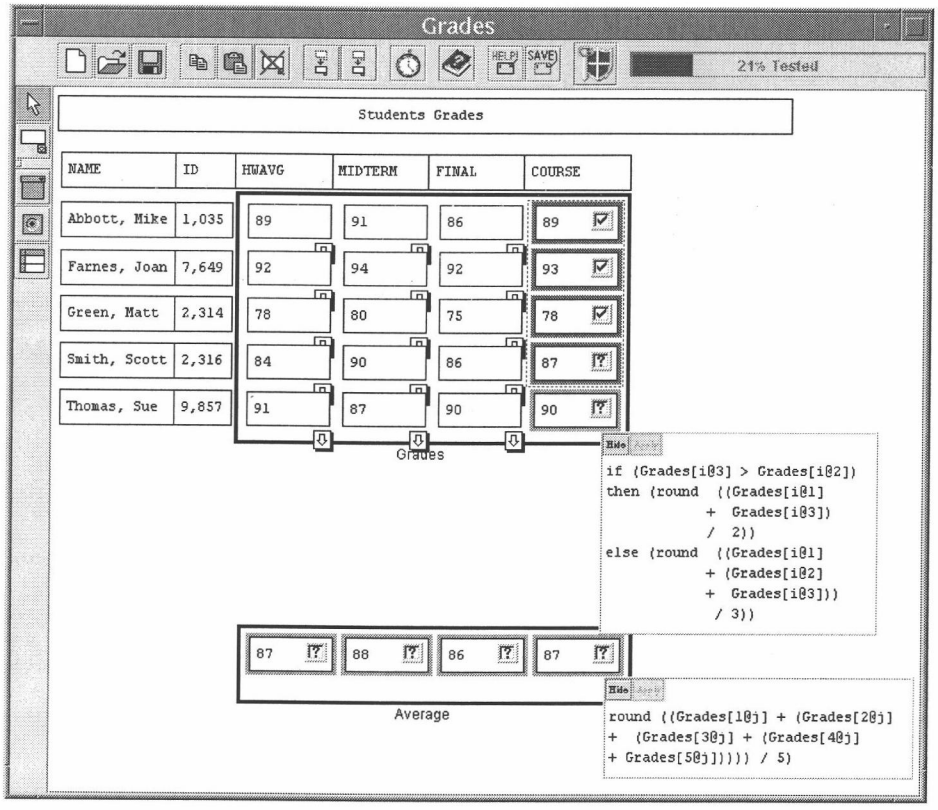


Figure 3. A version of the Grades spreadsheet using Forms/3 grids under the Straightforward approach. The user can enter a formula via a formula tab (⇩). The input cells each have their own formulas (one cell per region), but note that the rightmost column (region) has a single shared formula, as does the *Average* grid. The user is in the process of selecting four *COURSE* cells by stretching the dotted rectangle.

In previous work, other members of our research group have partially developed two approaches to testing large grids in spreadsheet grids [Sheretov 2000, Burnett et al. 1999]. The first approach was a straightforward extension of a WYSIWYT methodology for individual cells (and hence is termed the Straightforward approach). In the second approach, termed the Region Representative approach, the homogeneous cells shared most of the testing data, which added scalability by taking advantages of homogeneity.

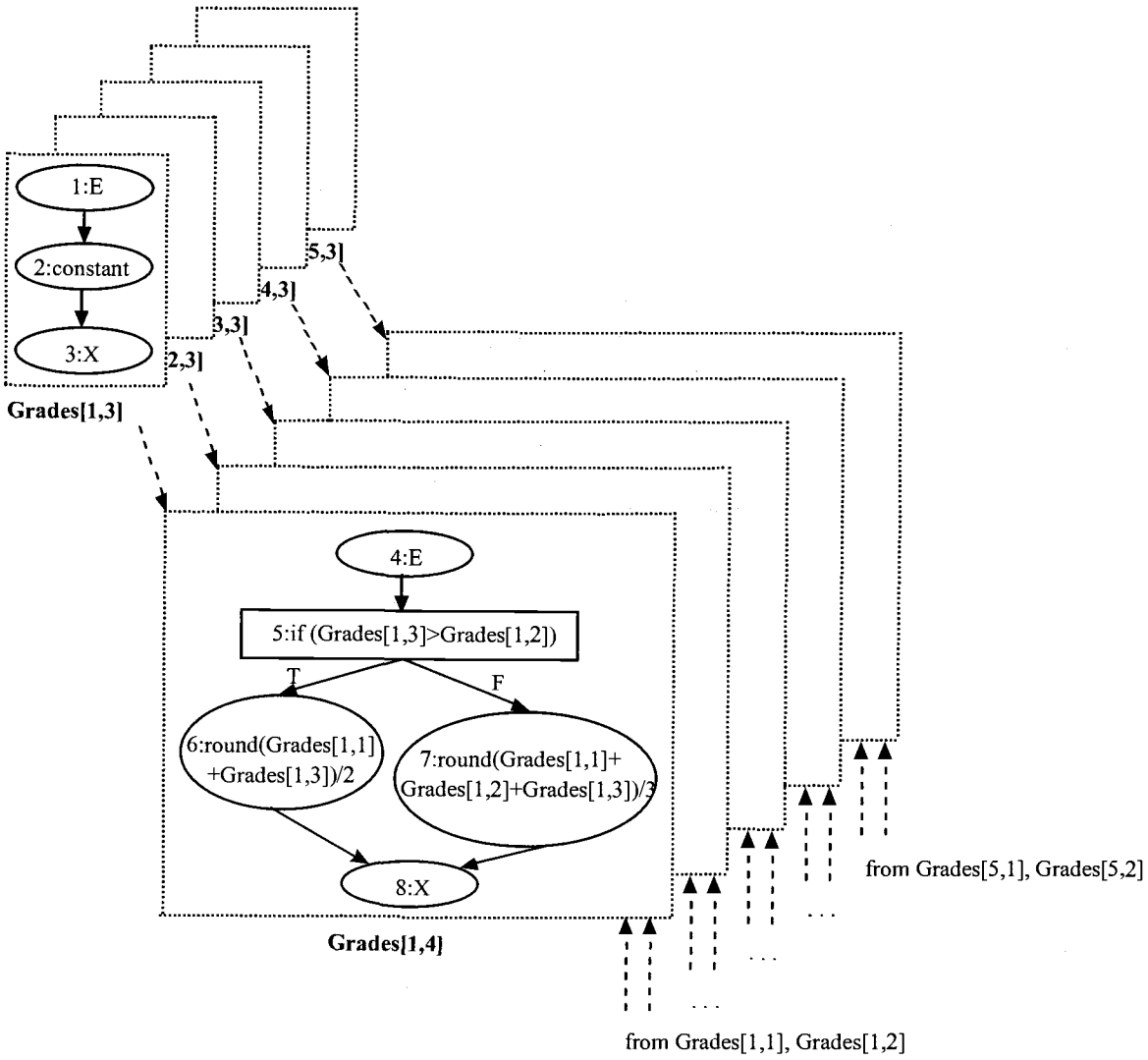


Figure 4. A portion of the CRG for the spreadsheet of Figure 3. Shown are formula graphs for the Grades grid.

2.4.1 Straightforward approach

One obvious approach to explicitly supporting grid testing is to let the user validate all or part of an entire region in one operation, but to have the system maintain testedness information about each cell individually. In the Straightforward approach, the only change from the user's perspective is that a group selection device such as a rubberband is added as Figure 3 shows. The user can use this device to select a group of cells in a grid and validate any of the selected cells, which applies the validation to all the selected cells. Therubberband does not "declare" any permanent relationship; it is simply a transient selection device. When the user does not use the rubberband, the user's validation of one grid cell `X` applies to `X` and only `X`, just as in the previous methodology. Other than the rubberband, the visual communication devices and four testing tasks' algorithms behind the scene are exactly the same as in the original WYSIWYT methodology as Figure 4 shows.

2.4.2 Region Representative approach

The Region Representative approach is a more elaborate approach. It does most of its reasoning at the granularity of entire regions rather than at the granularity of individual cells. Instead of a formula graph for each cell in a region `R`, `R`'s cells are modeled by a single formula graph of a region representative cell `Rij` in that region, as in Figure 5. Further, there is a special region that includes all input cells. This special region collapses all input values into one shared definition without losing the "use" circumstance. Five data structure components corresponding to those presented in Section 2.3 are now stored for each region representative instead of for each cell: `Rij`.*DirectProducers*, `Rij`.*DirectConsumers*, `Rij`.*Defs*, `Rij`.Uses, and `Rij`.*DUAs*. Only one component is still stored for each cell: `C`.*Trace*. And thereafter, three tasks – Task 1, Task 3, Task 4 - are now counted as region-level, and only Task 2 – tracking execution - is still the same as in the Straightforward approach.

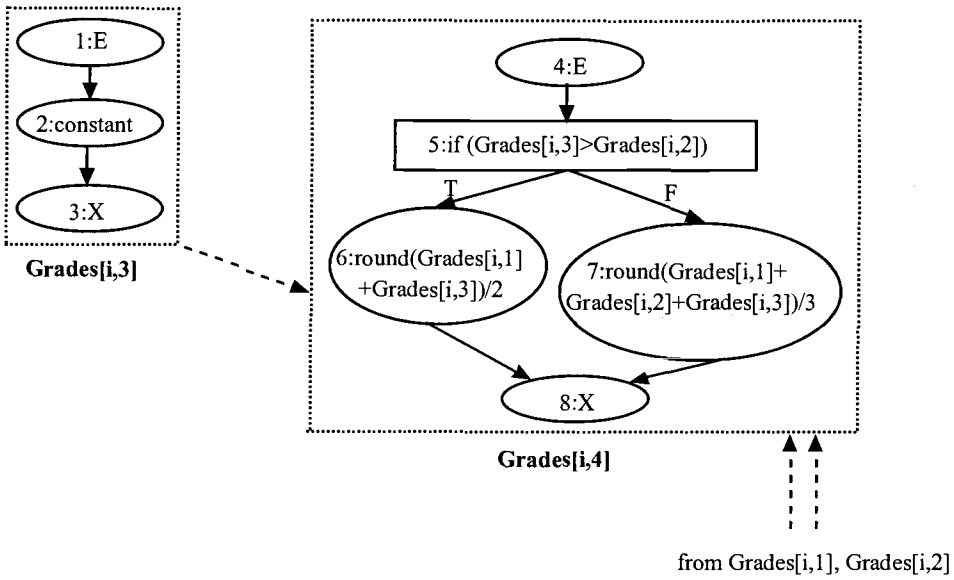


Figure 5. CRG showing the region representative of Figure 3's Grades' column 4, labeled *Grades[i,4]* here. Column 3 contains input cells that do not have any shared formulas, but form a constant region, labeled *Grades[i,3]*. Constant regions are also formed by columns 1 and 2. Note how much smaller this CRG is than the Straightforward approach's version of Figure 4: there are 4 CRG nodes in this figure, as compared to the 20 nodes in Figure 4.

In the previous research, the same visual device was used for the Region Representative approach as for the Straightforward approach. Even though in theory, the Region Representative approach could produce substantial time cost savings, the cost of updating visual devices compromised the effect. In this document, we improved the Region Representative approach by attending to sharing the costs of the visual devices so as to not lose the benefits of sharing the costs of behind-the-scenes reasoning about testing.

2.5 Recursive programs in spreadsheet languages

Although spreadsheet languages support end users, they also support programmers and some features are intended mainly for the latter audience. However, recursive programs in languages supporting end users can include programming structures that are rarely found in traditional languages, and this can complicate reasoning about testedness. For example, Figure 6 shows a solution to the *n*'th element of the Fibonacci sequence, which is the sum of the

previous two Fibonacci numbers. The solution involves three forms: one to compute the Fibonacci number for the desired *N* and two more to calculate the previous two Fibonacci numbers. (The notation *S:X* means a cell *X* on spreadsheet *S*. for example, *FIB:N* means a cell named *N* on a spreadsheet named *FIB*). The relationships are specified by the cell references, and the recursiveness lies in the fact that each form refers to yet another form, except in the base condition.

We term the original *FIB* the *model form*, and *56_FIB* and *64_FIB* *copy forms* of *FIB*. In Forms/3, copy forms inherit their model form's cells and formulas unless the user explicitly provides a different formula for a cell on a copy form. Changes in the model form are propagated to copy forms. To create such a program, the user first created a spreadsheet *FIB*, then copied it twice and got the copy forms *56_FIB* and *64_FIB*. The user changed cell *56_FIB:N*'s formula to *FIB:N-1* and cell *64_FIB:N*'s formula to *FIB:N-2*. Finally the user entered the formula for *FIB:ANSWER* and the system automatically created any other copies needed to calculate the results.

To support recursion as well as other uses of abstraction devices, Forms/3 uses an automatic generalization technique [Yang and Burnett 1994]. The purpose of generalization is to generalize the concrete forms for reuse. For spreadsheet languages based upon a single grid, such as commercial spreadsheet languages, generalization has been based strictly upon physical relationships. Forms/3 instead derives generality through the analysis of logical relationships and hence does not have the restrictions an ordinary spreadsheet language has, such as supporting only single grids. This, in turn, helps Forms/3 to succeed on advanced tasks such as recursion.

In Forms/3, generalization is lazy, being performed only when necessary and only on the cells requiring it. Without generalization, formulas such as the one for *56_FIB:Answer* in the copied spreadsheet in Figure 6 would be circular. This is one case that triggers generalization: when entry of a formula causes a cycle. There are also other cases which trigger generalization, for instance, saving a model form, making a new instance by copying a form, editing a copied cell that affects the generalized meaning of a previously generalized cell, or a cell or form being removed from the screen, etc.

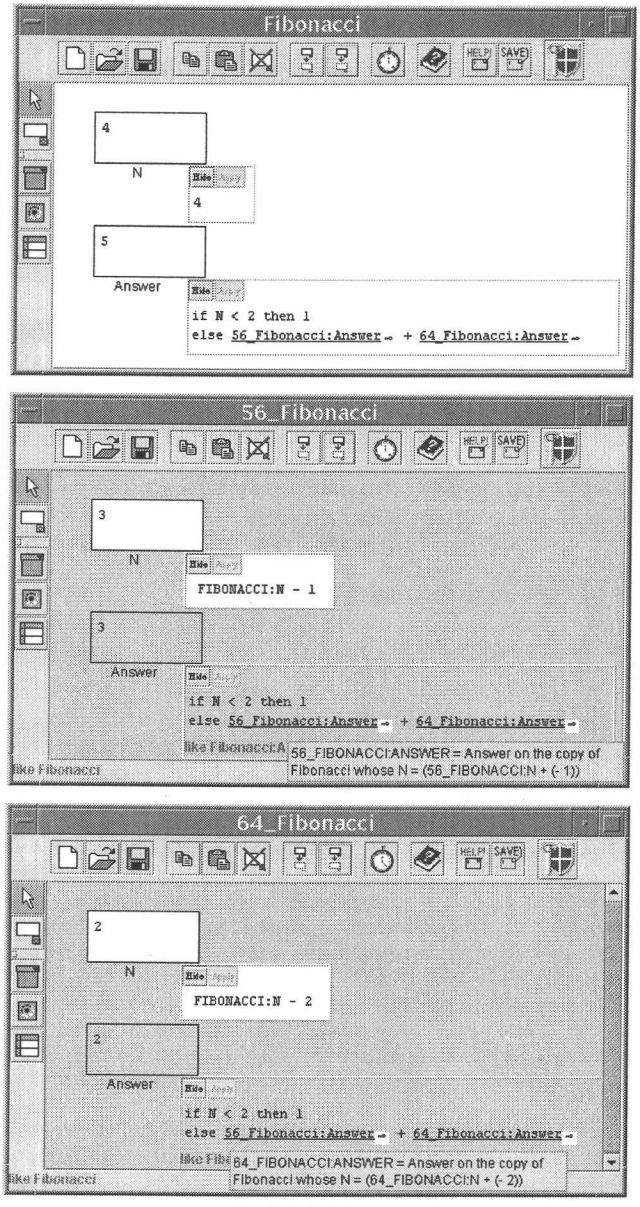


Figure 6. A recursive program to compute fibonacci number in Forms/3.

Through analysis of logical relations among cells, the generalized formula for *FIB:Answer* in Figure 6 becomes:

```
If    self:N < 2    then 1
Else  FIB(N ≡ self:N -1) :Answer
        +   FIB(N ≡ self:N -2) :Answer
```

For example, the second line means `Answer` on some copy FIB_i refers to cell `Answer` on whichever copy of `FIB` whose `N` is a reference to FIB_i's `N-1` cell. From the generalized notation, copied cells and their model can share the same generalized formula. At execution time, the system dynamically evaluates the formula to produce proper results. As shown in Figure 6, after generalization, concrete references to specific copies, which reflect the way the user entered them, are underlined to indicate that they are just samples. Moving the mouse over the underlined references displays a legend with the generalized reference for which the concrete version is a sample, as shown at the bottom of the figure.

2.6 User-defined abstract data types

Forms/3 supports user-defined abstract data types. Type definition forms are used to define new types. Type definition forms are similar to ordinary forms, but also contain a special type of cell called *abstraction box.* An abstraction box defines the composition of the type, and its value is an instance of the type. For example, Figure 7 defines a data type

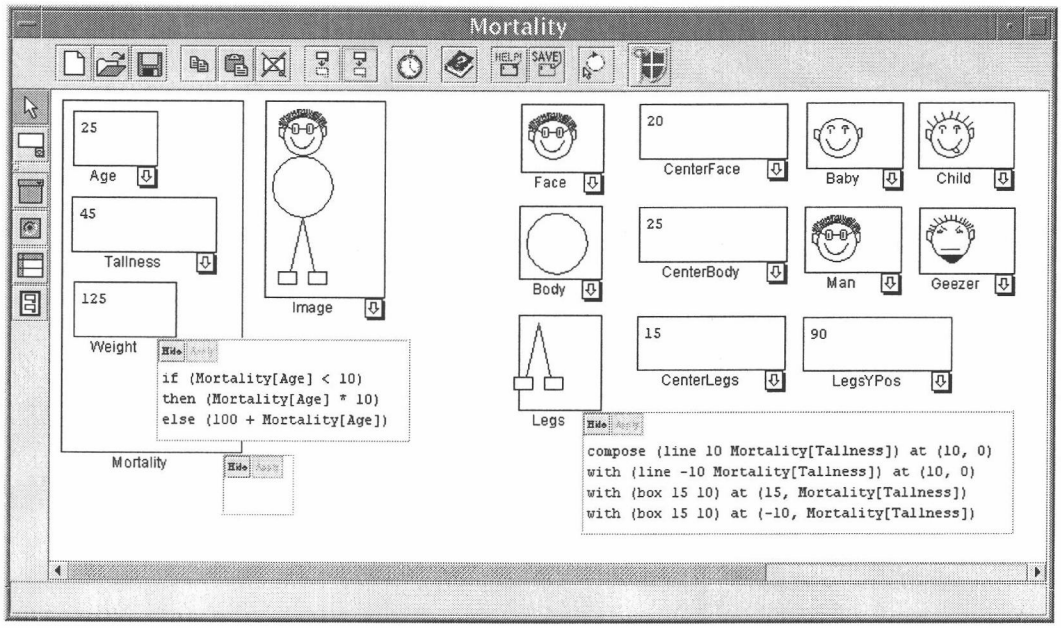


Figure 7. A user-defined abstract data type

Mortality. Using an abstraction box, the user has specified that the value of *Mortality*, an instance of this new type, consists of the *Age*, *Tallness*, and *Weight* cell's values. Because of this information, the implication is that the abstraction box's formula is simply the composition of the results of its interior cells.

Chapter 3: Completing the WYSIWYT Methodology for Spreadsheet Grids: Visual Devices Improvement[1]

There are three visual devices used in the original WYSIWYT methodology to communicate testedness to users about individual cells. Two of them—border colors and validation checkbox contents (checkmark, question mark, or blank)—have already been shown in the figures of Chapter 2. Here we consider the effects of these two devices and a third device on the time costs.

The third device available to users, which has not been shown in the figures to this point, is optional dataflow arrows[2] colored with "testedness" status in the same manner as the border colors. Not only do these arrows show dataflow paths among cells; when formulas are showing, they also show the interactions between formula subexpressions and the testedness of each—in other words, each du-association's testedness status. To users, this additional information seems to be almost as important as the border colors: in an empirical study of the original WYSIWYT methodology, 100% and 92% of the participants reported that the border colors and arrows respectively were helpful to their testing effort [Rothermel et al. 2000].

In the Straightforward approach, all three of the visual devices were employed unchanged for grids, just as described in Chapter 2. This was also the case in our earliest prototypes of the Region Representative approach. However, the impacts of this naive decision on both usefulness and time costs were dramatic. From the user's perspective, the number of arrows leading in and out of the cells in just a single grid were sometimes so great, the screen became swamped with these arrows, rendering worthless their communication value to users. From the system's perspective, the time savings accrued from the Region Representative approach's behind-the-scenes reasoning were so overshadowed by the high visual update costs, they were obliterated. Thus, both aspects of scalability were lost.

---

[1] The contents in this chapter and Chapter 4 are modified from sections of a paper "Testing Homogeneous Spreadsheet Grids with the "What You See Is What You Test" Methodology", co-authored with Margaret Burnett, Andrei Sheretov and Gregg Rothermel, *IEEE Transaction on Software Engineering* (to appear).

[2] To avoid adding too much clutter, each cell's arrows are transient, and appear/disappear when the user clicks on the cell.

For example, consider only cells `A` and `B` and grid `M1` in Figure 14. In the figure, there are two dataflow arrows pointing from `A` and `B` to each cell in `M1`. Clearly if `M1` were changed so that its last region contained 200 cells instead of just one, if the naive visual mechanisms

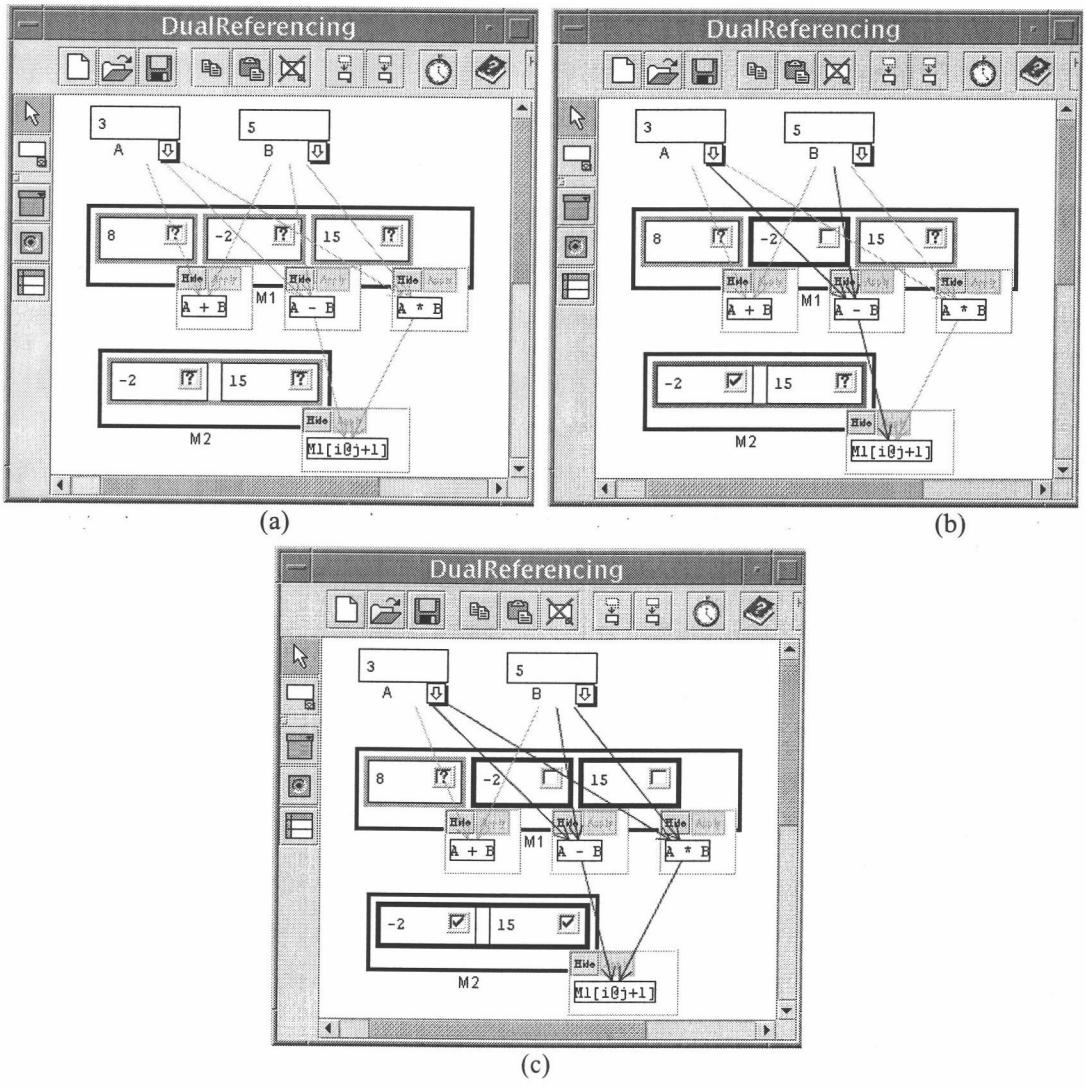


Figure 8. Grid user interfaces with testing border colors and dataflow arrows in Forms/3. Since the Region Representative approach reasons at the granularity of regions rather than cells, borders and arrows depict testedness statuses and relationships at the granularity of regions as well. (a) The initial state of spreadsheet DualReferencing with the optional arrows showing. (b) The user has validated M2[1,1]; hence several borders and arrows become more blue. (c) The user has now also validated M2[1,2], which turns more of the borders and arrows blue.

were employed, a click on `A` to reveal all of its arrows would swamp the screen with hundreds of arrows. In the worst case, all of these arrows would need to be repainted after formula edits and validation actions, as would the border colors around each of the 200 cells.

Changing the visual devices in the Straightforward approach to solve these problems did not seem reasonable, since the very essence of that approach is its reasoning about individual cells, and this must be reflected in visual communications with the user. However, the Region Representative approach reasons region by region rather than cell by cell, and we were able to reflect this reasoning granularity in the visual devices as Figure 9 shows. This can be seen by considering the rest of Figure 8. First, note that there is only one testing border, that surrounds the entire region in *M2*. Also, consider *M1[1,2]*'s definition, which *M2[1,1]* uses. This relationship is not depicted with arrows cell by cell, but rather region by region, by pointing into *M2*'s region's shared formula, or the region boundary if the formula is

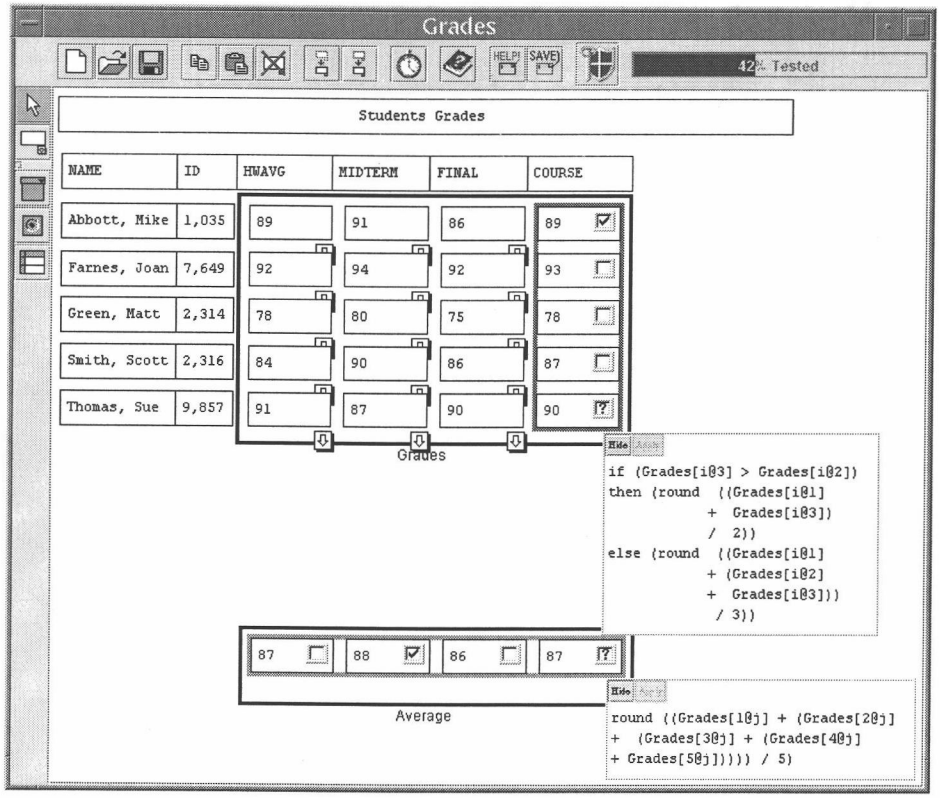


Figure 9. The *Grades* spreadsheet from Figure 3 shown here after visual device improvement of sharing border colors.

not showing. Thus, the arrows in the figure show that M2's single region uses only the rightmost two of M1's regions. In Figure 8(a), neither of these two relationships are tested, in Figure 8(b) the leftmost relationship is tested, and finally in Figure 8(c) the rightmost relationship is tested.

For a given spreadsheet *S*, let *r* be the number of regions in *S* and let *n* be the maximum number of element cells that any region has. Although Task 1 does not itself trigger visual updating, in order to support the later visual display of the colored arrows, Task 1 must collect du-associations about individual constant cells—in addition to the constant region representative du-associations that are sufficient for reasoning purposes. Doing so adds a cell-based loop to the Task 1 algorithm for the Region Representative approach, introducing a cost dependency on number of constant cells (*nr*) rather than on number of constant regions (*r*).

Task 3 and Task 4 entail visual updating, and these costs can be significant. Under the Straightforward approach, Task 3 requires repainting O(*rn*) cells to update each cell's testing border. If colored arrows are on display, the cost increases to $O((rn)^2)$. However, under the Region Representative approach, this task requires only O(*r*) updates of the testing borders, or O(*rn*) if colored arrows are displayed. For Task 4, the relationship between the Straightforward approach and the Region Representative is the same in terms of *r* and *n* as for Task 3, for border and arrow updating. In addition, Task 4 requires repainting the validation checkboxes of multiple cells: O(*rn*) cells for the Straightforward approach, and O(*r*) for the Region Representative approach. The actual pixel repainting algorithms are parts of the user interface toolkit, and incur the usual expense of reasoning about which pixels are visible and "dirty". (Since such details are not relevant to this thesis and are not part of our work, we do not present the algorithms themselves.)

Chapter 4: Performance Experiments on Two Testing Approach for Spreadsheet Grids

To provide concrete information about how the scalability of the Region Representative approach compares to that of the Straightforward approach, we conducted performance experiments. The purpose of these experiments was to complement the analyses of time complexity of testing grids (see [Sheretov 2000]), which are about theoretically worst cases, with evidence about the approaches' actual time costs in practice for large and small spreadsheets. Also, to measure the extent to which the effects of visual updating actually impact performance, the costs associated with visual updating are broken out separately.

Since the main objective of the experiments was to investigate scalability, we chose to vary only the size of the spreadsheet, holding other variables (such as degree of homogeneity, internal formula complexity, and interrelationships among formulas) constant. Figure 10 shows the spreadsheet used. In the figure, cells for 10 students are shown. The experiments were run on 5 different versions of this spreadsheet, involving 1, 10, 100, 200, and 500 students.

The spreadsheet used is similar to the simplified grade computation examples shown earlier in this paper, but contains a collection of formulas reflective of some common grading policies, such as allowing extra credit, rewarding improvement, and discarding the lowest quiz grade. Also, for compactness on our screen shots and analyses, additional input cells that do not add materially to the computations, such as for student names and IDs, have been omitted. The spreadsheet computes a course letter grade (A, B, C, D, or F). The scores range from 0 to 100. The first two columns of `Mins` track the minimum of each pair of quiz scores, and the final column is the minimum of the first two columns. The total score is the sum of the average of the three highest quiz scores, the points awarded based on the extra credit score, and bonus points for improvement. The letter grade is A, B, C, or D if the total score is greater than or equal to 90, 80, 70, or 60 respectively and is an F if the total score is less than 60.

In our first performance experiment, all non-constant formulas in the spreadsheet were edited, thus triggering Tasks 1, 2, and 4. We term this the *formula-editing experiment.* In our

second performance experiment, we validated the minimum number of cells needed to achieve full coverage (Task 3). We term this the *validation experiment.*

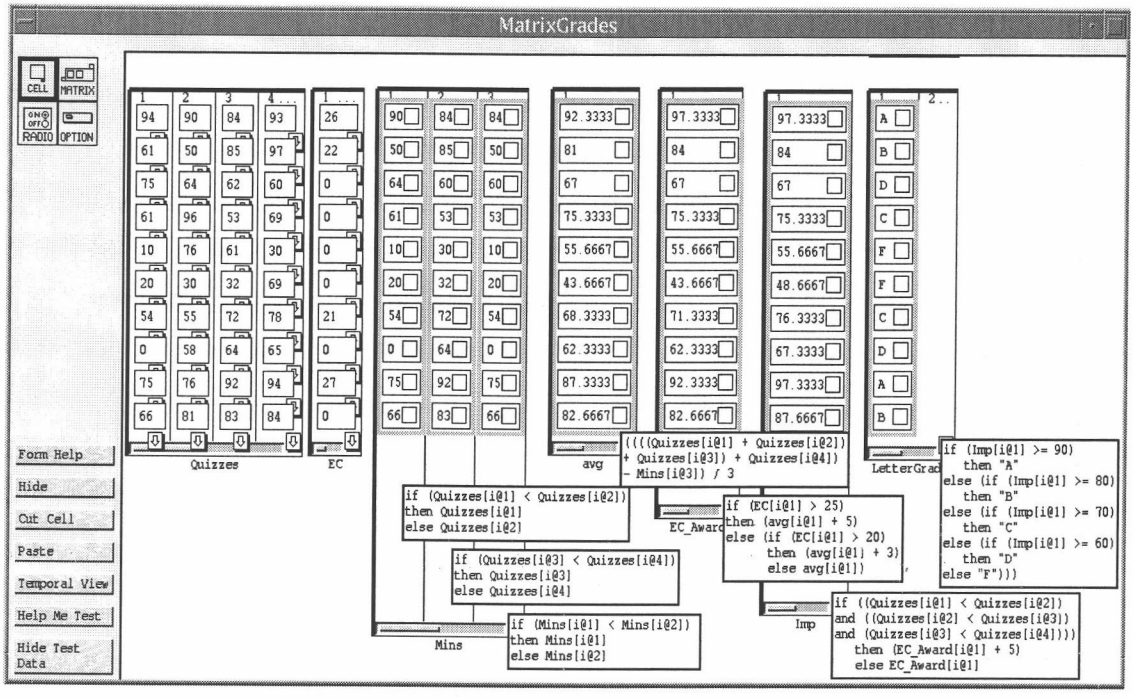


Figure 10. 10-student version of the `MatrixGrades` spreadsheet (using Garnet GUI). (The bottom borders have been arranged to allow the relevant formulas to display without overlap).

4.1 System-dependent performance comparisons

In order to compare runtimes of the two approaches, we ran both experiments on a Sun workstation and compared the timings of the Region Representative approach to those of the Straightforward approach. All timings are execution time averages of ten consecutive runs, and were taken on a Sun UltraSparc with 512 MB of RAM, with a single user, under Liquid Common Lisp 5.0.3 with Garnet [Myers et al. 1990]. Table 3 shows these comparisons for Task 1 in the formula-editing experiment, and the data is summarized in Figure 11 .

N-student version of `MatrixGrades` spreadsheet	Region Representative approach (seconds)	Straightforward approach (seconds)
1 student	0.037	0.297
10 students	0.072	0.526
100 students	0.442	3.827
200 students	0.851	9.865
500 students	2.012	57.364

Table 3. Execution times of Task 1 in the formula-editing experiment (collecting static information).

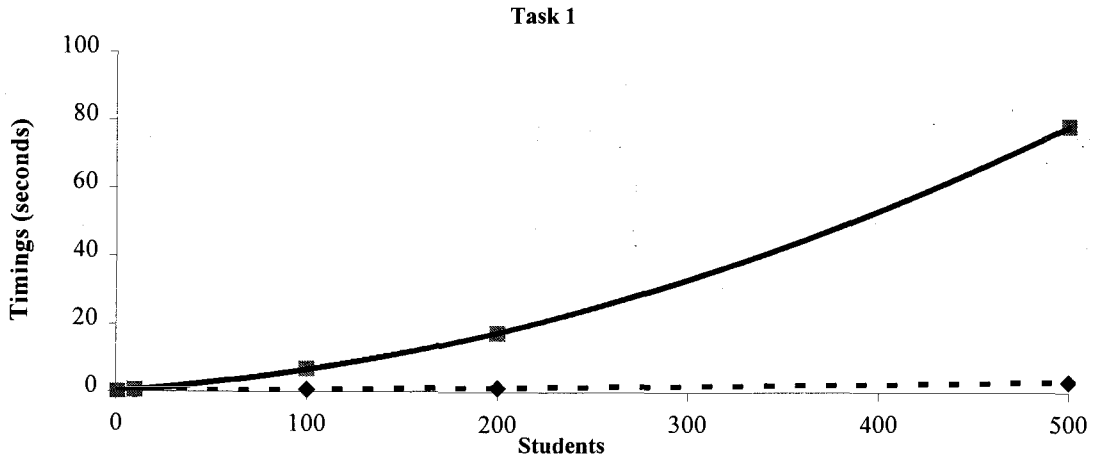


Figure 11. Comparison of the two approaches for Task 1. The solid line represents the timings under the Straightforward approach and the dashed line represents timings under the Region Representative approach. The Region Representative approach vertically overprints the Straightforward approach at 1 student and again at 10 students because the y-positions at those two x-positions are similar under both approaches.

The cost of Task 2 is ignored, because its costs were negligible and our measurement tools detected no difference between its performance under the two approaches.

Table 4 and Figure 12 display performance comparisons of the two approaches for Task 3, gathered in the validation experiment. In the experiment, the goal was to validate all cells in the spreadsheet. In this spreadsheet, under the Region Representative approach, it was possible, by carefully selecting test cases, to achieve 100% coverage in 10 validations. In fact, at least 10 is always required for this spreadsheet, which can be seen from the two rightmost formulas alone: `LetterGrade` has 5 cases and `Imp` has 2 cases, each of which interacts with each `LetterGrades` case, for a total of 10. In this spreadsheet, 10 is the maximum number of interactions among direct cell references, and the same inputs that exercise these were also chosen so that the other interactions were exercised at the same time. Given this configuration, under the Region Representative approach, we validated each cell in the `LetterGrade` column once, which achieved full coverage. To achieve full coverage in the Straightforward approach, each cell had to be validated 10 times, and the execution times for these 10 validations were summed up. Task 3 incurs visual updating, and its costs are detailed separately in the table.

N-student version of `MatrixGrades` spreadsheet	Region Representative approach (seconds)			Straightforward approach (seconds)		
	Without GUI update	GUI update	Total	Without GUI update	GUI update	Total
1 student[3]	0.010	0.010	0.020	0.010	0.010	0.020
10 students	0.136	0.378	0.514	0.925	0.736	1.661
100 students	0.155	2.011	2.166	10.195	28.128	38.323
200 students	0.188	3.652	3.840	20.837	72.345	93.182
500 students	0.210	7.324	7.534	57.423	275.913	333.336

Table 4. Execution times of Task 3 in the validation experiment (validating the cells needed to turn all testing borders blue).

[3] In the 1-student version study, under both approaches, we measured timings by validating the only cell in the LetterGrade column, which did not achieve full coverage. However, its timing is shown to demonstrate that the overhead expense of both approaches did not swamp the costs in small grids.

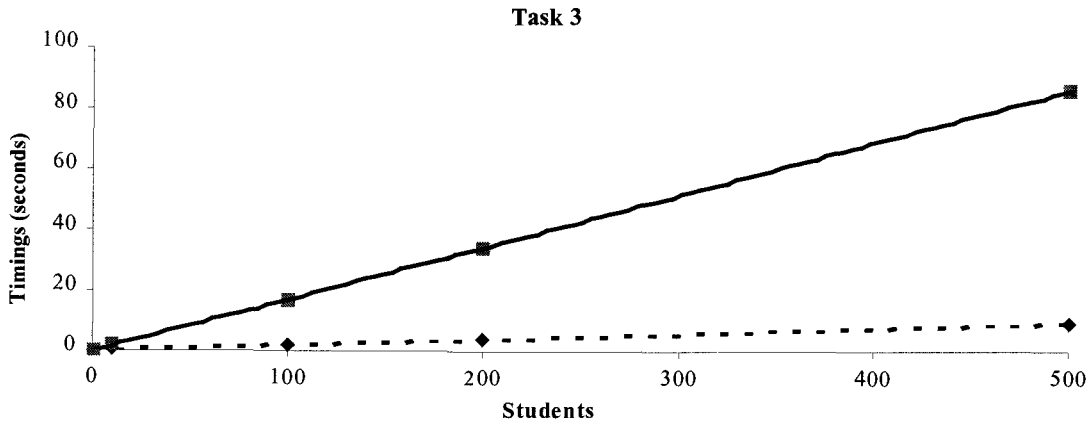


Figure 12. Comparison of the two approaches for Task 3 full coverage. To save space, the vertical axis is scaled more compactly than that in Figure 16 and Figure 18, but it is still sufficiently detailed to show the relationship between timings and number of students and to allow differentiating between the timing for 1 student versus the timing for 10 students.

Table 5 and Figure 13 present performance comparisons of the two approaches for Task 4, collected in the formula-editing experiment. As in the other tables, all timings are averages of ten consecutive runs. Like Task 3, Task 4 includes a visual component, which is detailed separately.

N-student version of `MatrixGrades` spreadsheet	Region Representative approach (seconds)			Straightforward approach (seconds)		
	Without GUI update	GUI update	Total	Without GUI update	GUI update	Total
1 student	0.010	0.052	0.063	0.010	0.026	0.036
10 students	0.010	0.135	0.145	0.039	0.416	0.455
100 students	0.010	0.679	0.689	0.311	3.342	3.653
200 students	0.010	1.211	1.221	0.528	6.158	6.686
500 students	0.010	2.678	2.688	1.356	19.953	21.309

Table 5. Execution times of the Task 4 in the formula-editing experiment (adjusting testedness information after editing non-constant formulas).

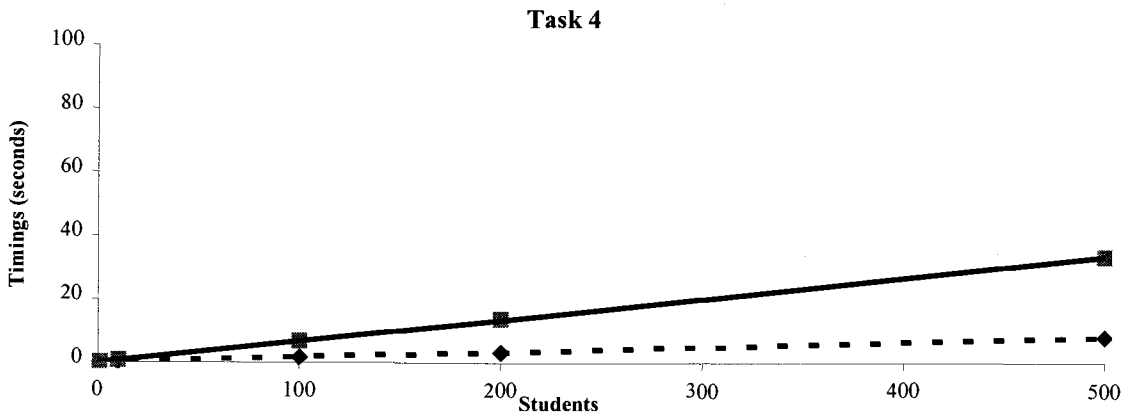


Figure 13. Comparison of the two approaches for Task 4, using the same scale as in Figure 16.

4.2 System-independent performance comparisons

In any system-dependent measure, performance results can be influenced by the particular configuration of the computer system being used. Thus, to understand whether comparisons on a particular system are representative in terms of the relative differences between approaches, it is also important to measure algorithms' major contributions in a system-independent way. Since our algorithms' final results are updated data structures, in this section we present system-independent measures of the algorithms by counting the numbers of these updates done in completing each task. These update counts were gathered as by-products of the same experiments that produced the results in the previous section. In addition to data structure updates by the system, actions by the user are another system-independent measure; hence, where there are differences in the number of user actions required to complete the task, we also report those differences here.

Table 6 presents a comparison of the two approaches for Task 1 in the formula-editing experiment. In Task 1, the user's goal is to edit a (non-constant) formula and the system's goal is to build formula graphs and to collect from them du-associations. The table counts the number of entries the system created in order to achieve this goal for all seven of the formula edits in the formula-editing experiment. (The amount of work the user performs in order to edit a formula is the same under both approaches.) The Region Representative approach's non-parenthesized numbers are those needed by the reasoning processes. In addition,

individual du-associations are needed for input cells to enable drawing of the colored arrows from the input cells individually, since they may not be physically located together and hence an arrow representing their constant region as a whole cannot be used here. These individual du-associations for input cells re-introduce some dependency on spreadsheet size: namely on the number of non-shared input cells present. Still, the creation of constant du-associations is less expensive than the creation of non-constant du-associations, which is why the runtime costs for this task increased very slowly as the number of cells increased (recall Table 3 and Figure 11).

N-student version of `MatrixGrades` spreadsheet	Region Representative approach		Straightforward approach	
	Formula graph nodes created	DU-associations created	Formula graph nodes created	DU-associations created
1 student	32 (32)	45 (45)	32	45
10 students	32 (77)	45 (207)	320	450
100 students	32 (527)	45 (1827)	3200	4500
200 students	32 (1027)	45 (3627)	6400	9000
500 students	32 (2527)	45 (9027)	16000	22500

Table 6. System-independent measures for Task 1.

In Task 3, the user's goal is to validate one or more values, providing new inputs if necessary by editing constant formulas, and the system's goal is to mark as exercised the du-associations involved in the approved value. Recall that, in the validation experiment, we set the number of values being validated to be the minimum number required to achieve 100% adequacy. Table 7 shows the number of updates done by the system to perform Task 3 in the validation experiment. As with Task 1, even in the Region Representative approach, individual constant du-associations (shown in parentheses) must be updated in order to support colored arrows rooted at the input cells.

N-student version of `MatrixGrades` spreadsheet	Region Representative approach (DU-associations marked exercised)	Straightforward approach (DU-associations marked exercised)
1 student	45 (45)	45
10 students	45 (207)	450
100 students	45 (1827)	4500
200 students	45 (3627)	9000
500 students	45 (9027)	22500

Table 7. System actions required to validate the MatrixGrades spreadsheet (Task 3).

Table 8 shows the range of user action counts required to perform Task 3 on the `MatrixGrades` spreadsheet in the validation experiment, assuming that the quiz and extra credit scores had previously been entered. In the table, the user edit actions under the Region Representative approach are 0 in the best case, which occurs if there are at least 10 students and if these students' inputs exercise all the du-associations. The probability of this being true increases with the number of students in the spreadsheet. In the worst case, the user must edit scores for 9 of the students in the grid in order to get the additional coverage needed beyond that provided by the real scores. Given the necessary input actions, 10 validations are required, one for each unique du-association path, to achieve 100% coverage. These counts follow the validation experiment's design, which count the minimum number of actions needed. Users could enter more inputs and validate more cells if they wished.

The user edit actions under the Straightforward approach come from the fact that each cell is reasoned about separately; hence to completely validate every cell, every cell must be forced through all 10 test cases. To achieve maximum coverage, each cell in the `LetterGrade` column will have to be validated 10 times. Because the user is allowed to select a set of cells and validate them with one click, the number of physical mouse clicks required to perform the validations could be reduced to as few as 10 validations if the user first rubberbands the entire group. However, there is no similar way to reduce the number of edit actions in the Straightforward approach: the user must enter enough inputs to force every student row through every one of the 10 cases. Such rubberbanding is best-case in terms of physical actions, but it does not save much cognitive activity: it would mean the user was

approving as many as 500 answers at a time, which implies that users need to scrutinize each of these 500 answers and pass judgment upon them individually, unless they work to reduce this by manually categorizing the answers into cases.

N-student version of `MatrixGrades` spreadsheet	Region Representative approach (actions)		Straightforward approach (actions)	
	Students whose scores must be edited	Validation actions	Students whose scores must be edited	Validation actions
1 student	9	10	9	10
10 students	0-9	10	90	10-100
100 students	0-9	10	900	10-1000
200 students	0-9	10	1800	10-2000
500 students	0-9	10	4500	10-5000

Table 8. User actions required to test the MatrixGrades spreadsheet (Task 3). The edit counts in the Straightforward approach are greater than the number of students because each student must be edited 9 times.

In Task 4, a user has just edited a (non-constant) formula and the system's goal is to remove its previous "exercised" marks on affected formula graphs and du-associations. Table 9 counts the number of updates the system made in order to achieve this goal in the formula-editing experiment. (The amount of work the user must perform in order to edit a formula is the same under both approaches.) The number of du-associations shown here are almost the same as for Task 3 because of the experiment's design, measuring the cost of exercising all the du-associations via Task 3 and resetting their statuses in Task 4. The only difference is that, since Task 4 does not involve changing constant du-associations' statuses, the constant du-associations from Table 8 are not applicable here.

N-student version of `MatrixGrades` spreadsheet	Region Representative approach (DU-associations updated)	Straightforward approach (DU-associations updated)
1 student	45	45
10 students	45	450
100 students	45	4500
200 students	45	9000
500 students	45	22500

Table 9. System-independent measures for Task 4.

4.3 Discussions

Both the Straightforward and the Region Representative approaches allow a *user* validation action on one cell to be leveraged across an entire region. This reduces user actions and, in the case of the Region Representative approach, also reduces manual test case generation. However, the user action savings available under the Straightforward approach are not as great as those available under the Region Representative approach, as the system-independent performance comparisons showed. Also, unlike the Straightforward approach, the Region Representative approach reduces the *system* time required to maintain testedness data, so that it removes much of the dependency of system time on grid region size. In keeping with this fact, the performance comparisons showed that the Region Representative's time savings increased as the size of the spreadsheet grew. This is critical in maintaining the high responsiveness that is expected in spreadsheet languages even in the presence of large grids. However, the performance experiments also demonstrated the tension between the system completing the tasks speedily in order to maintain responsiveness in the highly interactive world of spreadsheets versus providing as much immediate visual feedback about testedness as possible, which slows down the system.

Chapter 5: Visually Testing Recursive Programs[4]

As described in Section 2 and the previous two sections, researchers in our group developed the WYSIWYT methodology for testing individual spreadsheet cells, later extended it to large grids in which some cells share the same formula, and empirically validated its usefulness to both programmers and end users [Krishna et al. 2001, Rothermel et al. 2000]. However, recursive programs had not been supported by this methodology. It is important to support even these more powerful features, not just the "easy parts," to support the professional programmers end of the spectrum. In this section, we extend the WYSIWYT methodology to support recursive spreadsheets and linked copied spreadsheets in general.

5.1 Related work

We have not been able to locate any previous work regarding testing recursive programs for spreadsheet languages or for visual languages in general. In the larger software engineering community, most previous research regarding testing methodologies has been done in the context of traditional imperative languages, but even in that research community, testing techniques have paid little specific attention to whether or not a program is recursive. Testing based on code-based test adequacy criteria has potential to be cognizant of recursion, but in many cases these criteria are defined in a manner that renders them orthogonal to whether or not programs involve any recursive calls. For example, the statement coverage criterion requires that testers exercise each executable statement in a program at least once, but there is no requirement that any statement be exercised in any particular interaction among procedure calls.

---

[4] This chapter's content is a modified version of a paper with the same title, co-authored with Margaret Burnett, Andrew Ko, Curtis Cook and Gregg Rothermel, *HCC '01 2001 IEEE Symposium on Human-Centric Computing Languages and Environments* (to appear).

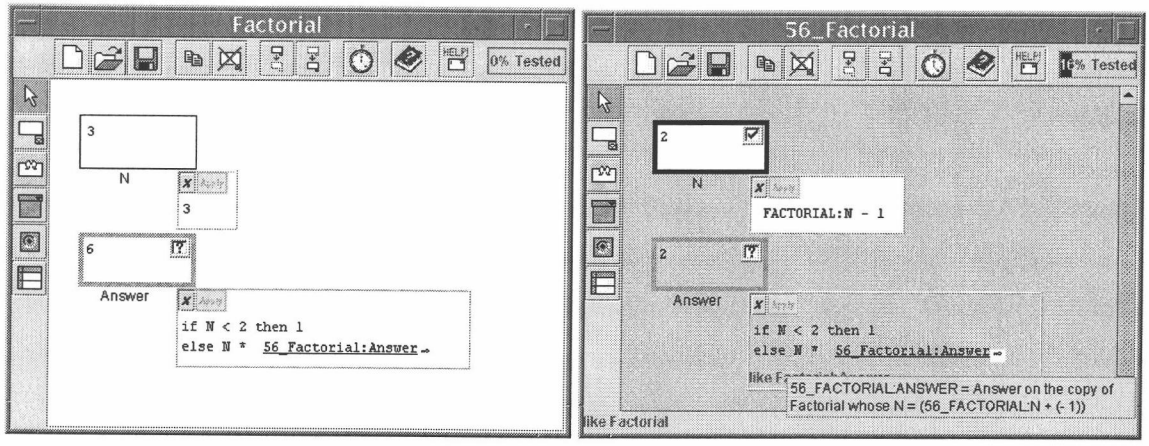


Figure 14. Forms/3 Factorial recursive spreadsheets.

Some test adequacy criteria, however, do consider interactions among procedures. Dataflow test adequacy criteria are among these: Dataflow criteria test interactions between definitions and uses of variables (du-associations), which can involve definitions and uses in distinct procedures, and cases where definitions reach uses across procedure boundaries. Some interprocedural data dependence analysis techniques [Harrold and Soffa 1994, Pande et al. 1994] specifically calculate interprocedural du-associations. For such techniques, given a du-association, the all-uses dataflow test adequacy criterion [Rapps and Weyuker 1985] (the criterion most closely analogous to ours) calculates cases where definitions can (statically) reach uses. However, it considers all cases where definition *d* reaches use *u* as a single equivalence class, even though *d* may reach *u* by multiple paths. Thus, for such criteria, whether *d* is defined and used in invocations k and k-1 of recursive procedure P or in invocations k+1 and k of recursive procedure P is immaterial: one "instance" of that du-pair must be exercised. In part, this stems from the underlying mechanism for calculating du-associations, which involves propagating definitions around a graph of the program, in which each procedure is represented by a single subgraph, regardless of the contexts in which it may be invoked. In contrast to this, the two approaches presented in this paper both take context into account.

5.2 Extended WYSIWYT approach

To test spreadsheets with recursion, one possible approach is a seemingly straightforward extension of the above testing methodology for non-recursive spreadsheets, maintaining testedness information about each cell individually based upon its dataflow relationships. We term this approach the "Extended WYSIWYT" approach. As described before in Section 2.5, Figure 14 is an example of a recursive spreadsheet, because a copy has been made of Factorial, and the original refers to the copy for some of its calculations.

In the Extended WYSIWYT approach, information on recursive copies is kept individually for each copied cell, and the user thus has the flexibility to validate any cell on any spreadsheet or copy. This has the advantage of being completely consistent with the way the WYSIWYT approach works on non-recursive spreadsheets.

5.2.1. Testing tasks and a problem with Task 1

To maintain and make use of testedness information, as described in Section 2.3, the system's work is divided into four tasks, each of which is triggered by a user action. The algorithms for Tasks 2-4 in the Extended WYSIWYT approach are the same as with non-recursive spreadsheets, but Task 1 requires further discussion.

In non-recursive spreadsheets, static du-associations are incrementally collected whenever the user edits cell `C`'s formula. The underlying assumption is that all cells that `C` references exist at the moment of a formula being accepted by the system; otherwise an error message would be produced and the formula rejected. However, for recursive spreadsheets, this assumption is not valid.

In Figure 14, when the user enters the formula of the cell `Factorial:Answer`, all cells that `Factorial:Answer` references exist, allowing the system to collect du-associations as usual. However, when the formula is copied to its copy `56_Factorial:Answer`, a problem arises: there is a reference to a cell named `Answer` (see legend at the bottom right) in another copy spreadsheet that has not yet been created by the system. Thus, for `56_Factorial:Answer`, there is not yet enough information for the system to collect all the static du-associations.

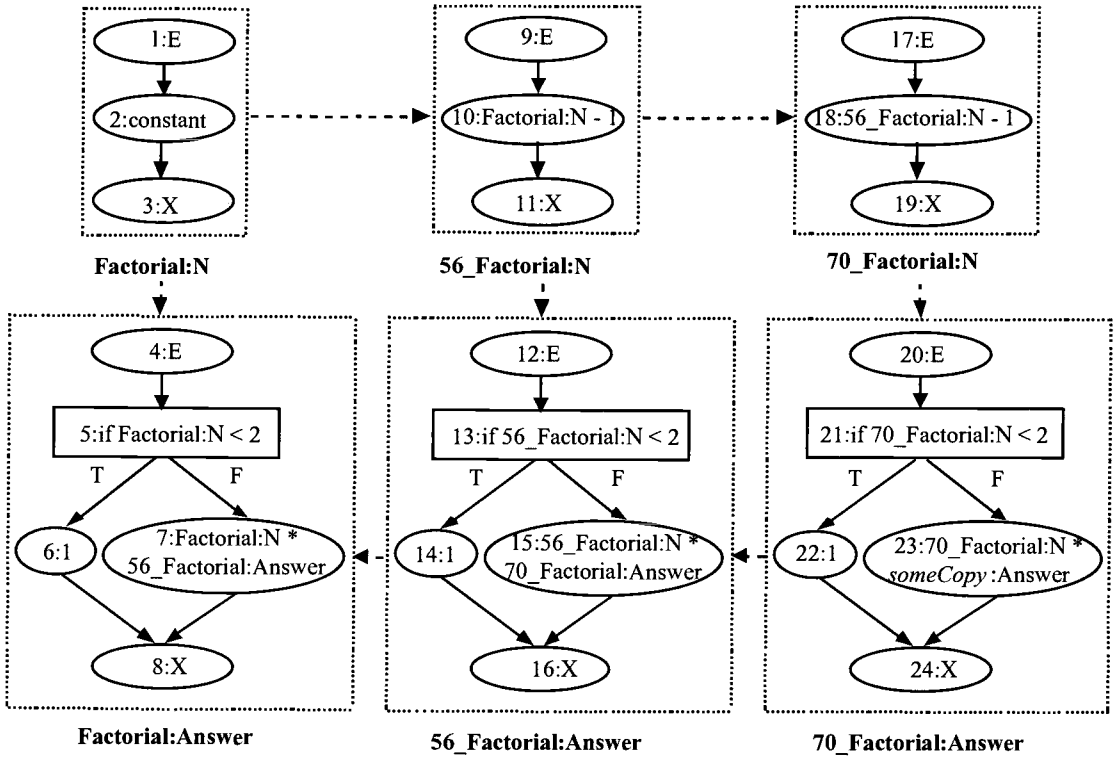


Figure 15. CRG for factorial recursive spreadsheets under the Extended WYSIWYT approach. Dashed arrows indicate dataflow relationships between cells. Within the formula graphs, *E* indicates entry into a formula and *X* indicates exit. Note, on the 70_Factorial:Answer, *someCopy*:Answer means the system doesn't have such a cell.

5.2.2 Algorithms to solve the problem.

To solve the problem, it is necessary to store some temporary du-associations as placeholders, thereby delaying the collection of some "static" du-associations until *after* the runtime evaluation of formulas reveals new static du-associations to collect. Doing so is necessary because both the original spreadsheet and its (modified) copy are visible and available to the user for viewing, editing, and referencing.

Algorithm *CollectIncomingAssoc* of Figure 16 is called when the user edits cell *c*'s formula or when a formula is copied to *c*. If *c*'s referenced cells exist, the system statically collects du-associations as usual; if not, the system builds temporary du-associations that are almost identical to the final ones. When the system eventually evaluates the formula, the

```
Algorithm CollectIncomingAssoc (C)
For each use ∈ C.Uses
  If a directProducer exists for use
    For each def ∈ directProducer.Defs
      Let DUA = (def, use, false)
      Add DUA to C.DUAs.Incoming
      Add DUA to directProducer.DUAs.Outgoing
  Else
    Let tempProducer be an identical copy of the directProducer
        referenced in use
    For each def ∈ tempProducer.Defs
      Let DUA = (def, use, false)
      Set DUA.temporary = true
      Add DUA to C.DUAs.Incoming
      Add DUA to tempProducer.DUAs.Outgoing
    Add use to C.TemporaryUseList
```

```
Algorithm RebuildAssoc (C)
For use ∈ C.TemporaryUseList
  If a directProducer exists for use
    Let temporaryDUAs = use.DUAs.Incoming
    DeleteDUAs ( temporaryDUAs)
    For each def ∈ directProducer.Defs
      Let DUA = (def, use, false)
      Let C be the cell containing use
      Add DUA to C.DUAs.Incoming
      Add DUA to DP.DUAs.Outging
    Remove use from C.TemporaryUseList
```

Figure 16. Algorithms of collecting incoming du-associations under the Extended WYSIWYT approach.

algorithm *RebuildAssoc* (Figure 16) is called. If, as a result of evaluation, the system created copies containing *use*'s missing references, the system replaces the temporary du-associations with real du-associations.

As with the versions of these algorithms for the original WYSIWYT approach [Rothermel et al. 2001], the time complexity of the algorithm *CollectIncomingAssoc* is of the same order as the evaluation engine's cell traversal: $O(p_d)$, where p_d is the number of cells directly referenced by a cell. The algorithm *RebuildAssoc* runs in time $O(mp_d)$, where m is the maximum number of definitions in a cell's formula. In practice, m is small and constant-bound (in Factorial, it is 1) due to the fact that, for most spreadsheet languages, there is a maximum

imposed on formula lengths. So the algorithm *RebuildAssoc* adds little overhead to the work needed at evaluation time.

5.2.3 Exacerbated infeasible du-associations

As described in Section 2.3, even without recursion, it is not always possible to test all du-associations, but we need to avoid exacerbating the problem. Unfortunately, the Extended WYSIWYT approach does exacerbate the problem, because the temporary du-associations solution (Section 5.2.3) is not sufficient for cells on spreadsheet copies that compute base conditions. For example, in Figure 17, `70_Factorial:Answer`'s reference to another copy of Factorial will never be created or executed because `70_Factorial:Answer` computes the base (See Figure 14).

Even if the user decides to change `N`'s formula to a large input value, the problem does not go away. Although it is solved for `70_Factorial` in that the `70_Factorial` spreadsheet is not the base any more, it re-materializes for whatever new copies the system automatically creates to compute the base condition. Thus there will always be an infinite number of static du-associations under the Extended WYSIWYT approach, and a user's goal of 100% testedness of all viewable spreadsheets is unattainable.

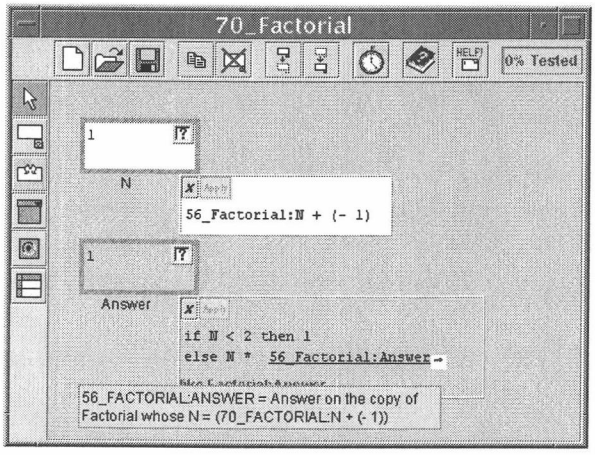


Figure 17. The base condition of Figure 14. `70_Factorial` is the copy of the spreadsheet in Figure 14 that computes the base condition.

5.3 Copy Representative approach

The Copy Representative approach was inspired by the Region Representative approach as described in Section 2.4, which was developed to remove the tedium of testing groups of cells with shared formulas [Burnett et al. 1999]. The idea behind the Region Representative approach was to share most of the testing data among these similar cells.

As described in Section 2.5, in Forms/3's generalization model [Burnett et al. 2001], when the user enters a formula for a model cell, the system will automatically produce a generalized version, which is shared among the model and its copies. When doing recursive programs, the system dynamically evaluates the generalized formula and creates necessary copies of forms, as references are made to cells on them. Thus the idea behind the Copy Representative approach is to allow these copies also to share testing data. A potential advantage is that users might think it more logical for copies to share testing data.

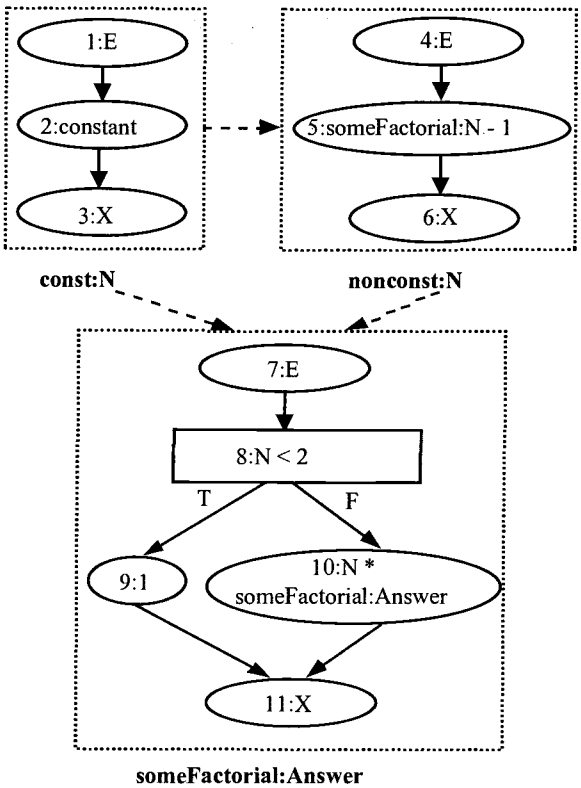


Figure 18. CRG for factorial recursive spreadsheets under the Copy Representative approach.

5.3.1 Changes to the CRG Model

In the Copy Representative approach, when cells have the same formula, they share a single formula graph. Thus, instead of building a concrete formula graph for every cell, as in the Extended WYSIWYT approach, the system builds a generalized formula graph for the model cell and its unedited copies, as in Figure 18. Further, as with the Region Representative approach, all copies of input cells (with constant formulas) are represented by a single formula graph, whether edited or not.

The CRG in this figure is much smaller than the CRG in Figure 15. This difference carries significant implications for the user's interaction: a validation of one cell *C* now propagates to every other cell relating to the same model, even though in both approaches, users see the same visual device. For example, if the user validated `Factorial:Answer`, all `Answer` cells in `Factorial`'s copies would also be validated.

5.3.2 Information collected

To realize the Copy Representative approach, most of the testing-oriented information corresponding to that described in Table 2 of Section 2.3, must be shared among shared cells; see Table 10. As the table shows, we introduced a *SharedGraph* (in the implementation, this is called *CFGInfo*), which is the shared formula graph among the model and its unedited copied cells. Cells no longer have a one-to-one relationship with a formula graph; instead, there is a many-to-one relationship between cells and their shared formula graph. The *SharedGraph* keeps most testing information, such as generalized formulas, definitions, uses, and du-associations as Table 10 shows.

	Information collected	Description
1 - 2	*C.DirectProducers* *C. DirectConsumers*	Same as Table 2, being kept in cell level
3 - 5	*SharedGraph.Defs* *SharedGraph.Uses* *SharedGraph.DUAs and its components*	Same as Table 2, except replacing "C" by "SharedGraph", which is the shared formula graph for cells with the same formula.
6	*C.Trace*	The set of shared formula graph nodes that were executed in the most recent evaluation of C.

Table 10. The primary testing information collected under the Copy Representative approach. The row numbers 1-6 correspond to the rows in Table 2 of Section 2.3 with differences from that table being underlined.

5.3.3 Initialization: when to build the shared formula graph (SharedGraph)

When a cell is created or edited, it must be given either a new or an existing formula graph. If a model or copied cell's formula has been edited directly by the user, the cell requires a new formula graph. Otherwise, the cell is an identical copy or a non-model *input cell* (an *input cell* is a cell with a constant formula), and can point to an existing formula graph as the algorithm in Figure 19 shows.

```
algorithm BuildSharedGraph ( C )
  If C is a model cell or an overridden cell
    Initialize aSharedGraph to empty
    CollectAssoc (C, aSharedGraph)
  Else if C is an non-overridden copy cell
          or C is a non-model input cell
    Let model = C's model cell
    Let aSharedGraph = model.SharedGraph
  Let C.SharedGraph = aSharedGraph   // remainder: this is a pointer
```

Figure 19. Algorithm of initialization of testing tasks. AlgorithmbuildSharedGraph statically determines whether to build a new shared formula graph or to point to an existing shared formula graph.

```
Algorithm CollectAssoc (C, SharedGraph)
Let oldSharedGraph = C.SharedGraph
// (1) collecting incoming information
For each use ∈ SharedGraph.Uses
  Let useCells = {copies sharing SharedGraph}
  Let allDefCells = { definition cells that contribute to useCells }
  Let defSharedGraphs = {defCell.SharedGraph | defCell ∈
allDefCells}
  For each defSharedGraph ∈ defSharedGraphs
    For each def ∈ defSharedGraph.Defs
      Let DUA = (def, use, false)
      Add DUA to SharedGraph.DUAs.Incoming
      Add DUA to defSharedGraph.DUAs.Outgoing
// (2) collecting outgoing information
For each def ∈ SharedGraph.Defs
  For each DUA' ∈ oldSharedGraph.DUAs.outgoing
      Let use = DUA'.Use
      Let useSharedGraph be the SharedGraph containing use
      Let DUA = (def, use, false)
      Add DUA to SharedGraph.DUAs.Outgoing
      Add DUA to useSharedGraph.DUAs.Incoming
DeleteSharedGraphInformation(oldSharedGraph)
```

Figure 20. Algorithm of collecting du-associations under the Copy Representative approach.

5.3.4 Task 1 reasoning: collecting static du-associations

Du-associations among generalized representatives of cells, not among cells themselves, are collected statically, and are later resolved dynamically to the concrete cells they represent. For example, in Figure 18, the top left formula graph represents all N cells with constant formulas. The top right formula graph represents all the other N cells, which have a single shared (non-constant) formula. Similarly, the bottom formula graph represents cell `Answer` on all copies of `Factorial`.

Algorithm *CollectAssoc* in Figure 20 uses a shared formula graph (*SharedGraph*) for all copies instead of collecting static du-associations for each concrete cell. When a user edits a cell, the cell and its *SharedGraph* are passed to algorithm *CollectAssoc*. To collect incoming du-associations of *SharedGraph*, for each generalized use cell sharing this formula graph

(possibly on multiple spreadsheet copies), the algorithm considers all definition cells which contribute to the use. From these definition cells, the system can build du-associations between the definition cell's shared formula graphs, and this *SharedGraph*. To collect outgoing du-associations of *SharedGraph*, the algorithm utilizes the old *DUAs* of *SharedGraph*. For each use which is the use node of the DUA $\in$ *DUAs.outgoing*, the system can get the set of use cell's shared formula graphs which contain the use, and then build du-associations between the *SharedGraph* and the use cells' shared formula graphs. Finally, the system deletes prior information. With this algorithm, the system does not need the temporary du-associations of the Extended WYSIWYT approach.

5.3.4.1 Time complexity of Task 1

CollectAssoc is a modified version of an algorithm developed for the original WYSIWYT methodology [Rothermel et al. 2001] for replacing concrete cells with shared formula graphs. To collect incoming du-associations, there are three loops. Let f be the number of reference in the formula of *SharedGraph* in the outer loop; let p_d be the number of shared formula graphs which are potentially referenced by *SharedGraph*'s uses in the second loop, corresponding to concrete cells' direct producers; let h be the number of definitions in *SharedGraph* in the innermost loop. To collect outgoing du-associations, there are two loops. Let g be the number of definitions in *SharedGraph* in the outer loop; and let c_d be the number of shared formula graphs which use *SharedGraph*'s definitions in their formula in the inner loop, corresponding to concrete cells' direct consumers. The total time cost of the algorithm is:

$O(fp_dh + gc_d)$

The above cost can be further simplified when there is a maximum imposed on formula lengths. Most spreadsheet languages have such a maximum; for example, in Excel, the maximum is 1024 characters. Given the presence of such a maximum, f, h and g become constant-bounded by the maximum formula length. Thus the simplified asymptotic time cost of Task 1 is:

$O(p_d + c_d)$

Which is the same as the original WYSIWYT methodology cost to collect du-associations for one cell [Rothermel et al. 2001]. Our system collects du-associations for one shared formula graph (1 or more concrete cells).

5.3.4.2 Cost of Task 1 in context

This algorithm is triggered when the user edits a model cell C's formula. At this point, the costs any spreadsheet system must incur even without the existence of a testing subsystem are those of parsing the formula, which costs at least the number of characters in the formula; of calculating at least the on-screen cell *C* and its copies and some of their producers; and of notifying consumers of the edited cell that their values are out of data, requiring recalculation and/or discarding of any previously cached values [Rothermel et al. 2001].

Of these three costs, the costs of evaluation and notification are the most useful to consider, because they are the greatest that involve the same cell sets as in Task 1. If a model cell and all its copied cells are on the screen, the cost of evaluation is at least as great as $O(p_d)$, because the system needs to revisit at least all direct producers and to recalculate producers (including direct producers) that do not already have up-to-date cached values. Notification of consumers requires the system to visit at least all the direct consumers. Since evaluation and notification visit at least the same cells as in Task 1, then when the model cell *C* and its copied cells are all on the screen, the cost of Task 1, in the context in which it is performed has the same order as the system evaluation and notification tasks.

5.3.5 Task 2 reasoning: collecting execution trace information

When a cell is executed, a trace of its execution is saved. Unlike the du-associations and formula graphs, the execution traces are not shared among cells under the Copy Representative approach, because different cells with the same formula may execute different parts of that formula. For example, in Figure 14, the cell `Factorial:Answer` executes the else-expression of the formula whereas Figure 17's `70_Factorial:Answer` executes the then-expression. Execution traces are collected via an O(1) probe in the evaluation engine.

5.3.6 Task 3 reasoning: marking DUA "covered" when the user validates a cell

Figure 21 gives the validation algorithm. The system gets the du-association of a cell reference from the shared formula graph, and then validates the du-association. The call to *dynamicallyResolve* finds the concrete cell represented in this du-association in the context of `C` and recursively validates further. This algorithm is slightly modified from the algorithm in the original WYSIWYT methodology, with the difference of a call to *dynamicallyResolve* to derive the cell `C`'s direct producer. Algorithm *dynamicallyResolve* is using Forms/3 automatic generalization technology, and is an O(1) routine that returns the actual cell which `C` references. For example, the cell `56_Factorial:Answer` in Figure 14 contains a reference to `someFactorial:Answer`, then *dynamicallyResolve*(`someFactorial:Answer`, `56_Factorial:Answer`) returns `70_Factorial:Answer` through dynamically resolving the generalized formula as described in Section 2.5.

The algorithm *ValidateCell* starts by traversing du-associations for all the direct producers used in `C`'s trace (the top two loops), and finally has a tail recursive call to the producers of these producers. Thus, the total time cost is simply:

O (p)

where p is the number of `C`'s direct and transitive producers.

Task 3 is triggered when the user performs one validation of cell `C`. Its cost is similar to the cost of evaluating a cell when there are no relevant cached values, but unlike the evaluator, Task 3 does not need to access any consumers to notify them that their cached

```
Algorithm ValidateCell (C)
Let aSharedGraph = C.SharedGraph
For each use ∈ C.Trace
  For each DUA ∈ aSharedGraph.DUAs.incoming do
    If DUA.use = use then
      Let defCellRef = DUA.def
      Let directProducer = dynamicallyResolve(defCellRef,C)
      DUA.exercised = true
      ValidateCell (directProducer)
```

Figure 21. Algorithm of updating testedness following a validation.

values are out of date.

Thus, the time cost of Task 3 for one cell is the same as that in the original WYSIWYT methodology [Rothermel et al. 2001].

5.3.7 Task 4 reasoning: resetting affected cells to "not covered" when the user edits a formula

Figure 22 provides the algorithm *AdjustTestedness*. For each outgoing du-association in the SharedGraph, which *C* points to, it marks the du-association "not covered" and recursively processes *C*'s direct consumers.

```
Algorithm AdjustTestedness (C, UnValidatedID)
Let aSharedGraph = C.SharedGraph
For each DUA ∈ aSharedGraph.DUAs.Outgoing
  DUA.exercised = false
  Let useCellRef = DUA.use
  If useCellRef.UnValidatedID < UnValidatedID then
     Let directConsumer = dynamicallyResolve(useCellRef, C)
     AdjustTestedness (directConsumer, UnValidatedID)
```

Figure 22. Algorithm of adjusting testedness when a shared formula is edited.

The explicit loop traverses the du-associations in direct consumers of *C*. There is also a recursive call to consumers of these consumers. The algorithm is similar to that of the original WYSIWYT methodology [Rothermel et al. 2001], except that it dynamically resolves each cell's concrete direct consumers. Thus the total time cost of Task 4 is:

O (c)

where c is the number of *C*'s consumers (including direct and indirect consumers). The time cost is the same as that in the original WYSIWYT methodology [Rothermel et al. 2001].

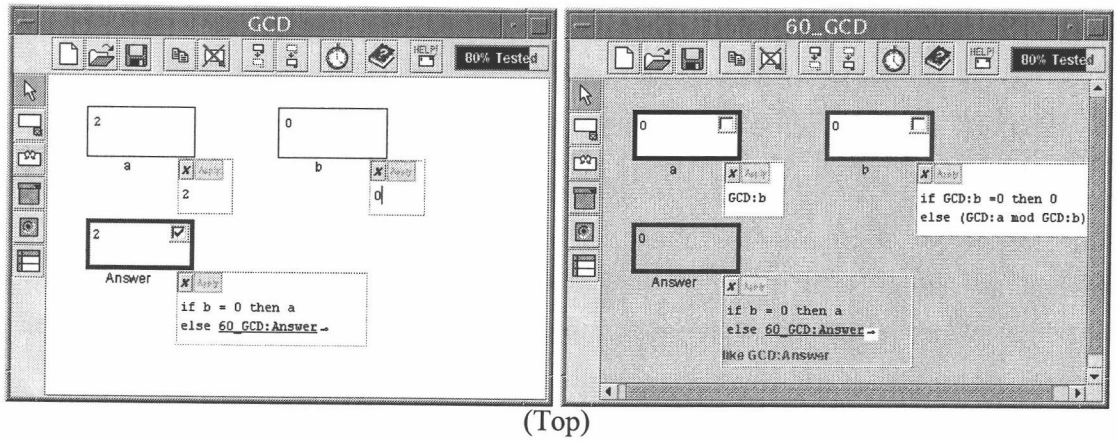
Task 4, like Task 1, is triggered when a new formula is entered for cell *C*. Most evaluation strategies require visits to all the consumers of *C* for purposes of discarding cached values and/or recalculating them, the specifics of which depend on whether the engine uses

lazy or eager evaluation. Because of this fact, Task 4 adds only O(1) to other work that is normally performed by a spreadsheet system without a testing subsystem.

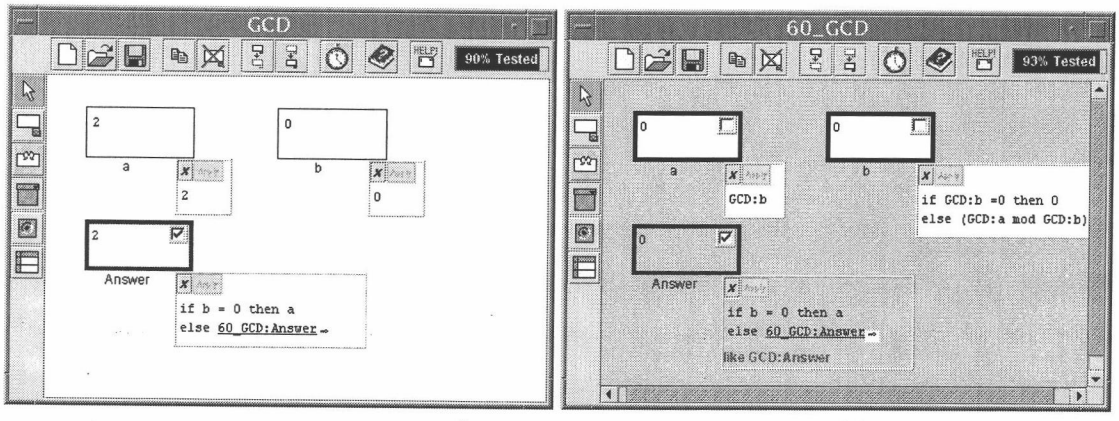
5.3.8 A variation: Model-only approach

In the design phase of our empirical study, people in our group suggested one more testing approach for the recursive programs, namely the "Model-only" approach. Under this suggestion, the data structures are the same as the Copy Representative approach, but the visual devices are different. The users can perform testing actions only on cells that are not copies. The arguments for this approach ask the question why users would care about copies if they have same formula as the model. In this approach, for example, as in Figure 23's top part, the user can validate only cells that are not copies and have a checkbox. Thus the user can only validate `GCD:Answer`, but can not validate `60_GCD:Answer`.

We found some flaws with this approach. The model-only approach may introduce some infeasible du-associations. For example, the spreadsheet GCD, as in Figure 23, if using the Model-only approach, will have two more infeasible du-associations as shown in Figure 24. Thus, users cannot reach as much testedness as users can under the Copy Representative approach. As shown in Figure 23, the testedness after all possible test cases is 80% under the Model-only approach – the top part of Figure 23 - versus the testedness after all possible test cases is 90% on the model form and 93% on the copied form under the Copy Representative approach – the bottom part of Figure 23. There is another potential flaw as well. The purpose of WYSIWYT is to give users powers of visually testing programs. However, under the Model-only approach, one of these powers is withheld from the users in that they are not able to check off values on copied cells. This partial removal of power to choose which cells to test may seem confusing or arbitrary to some users.

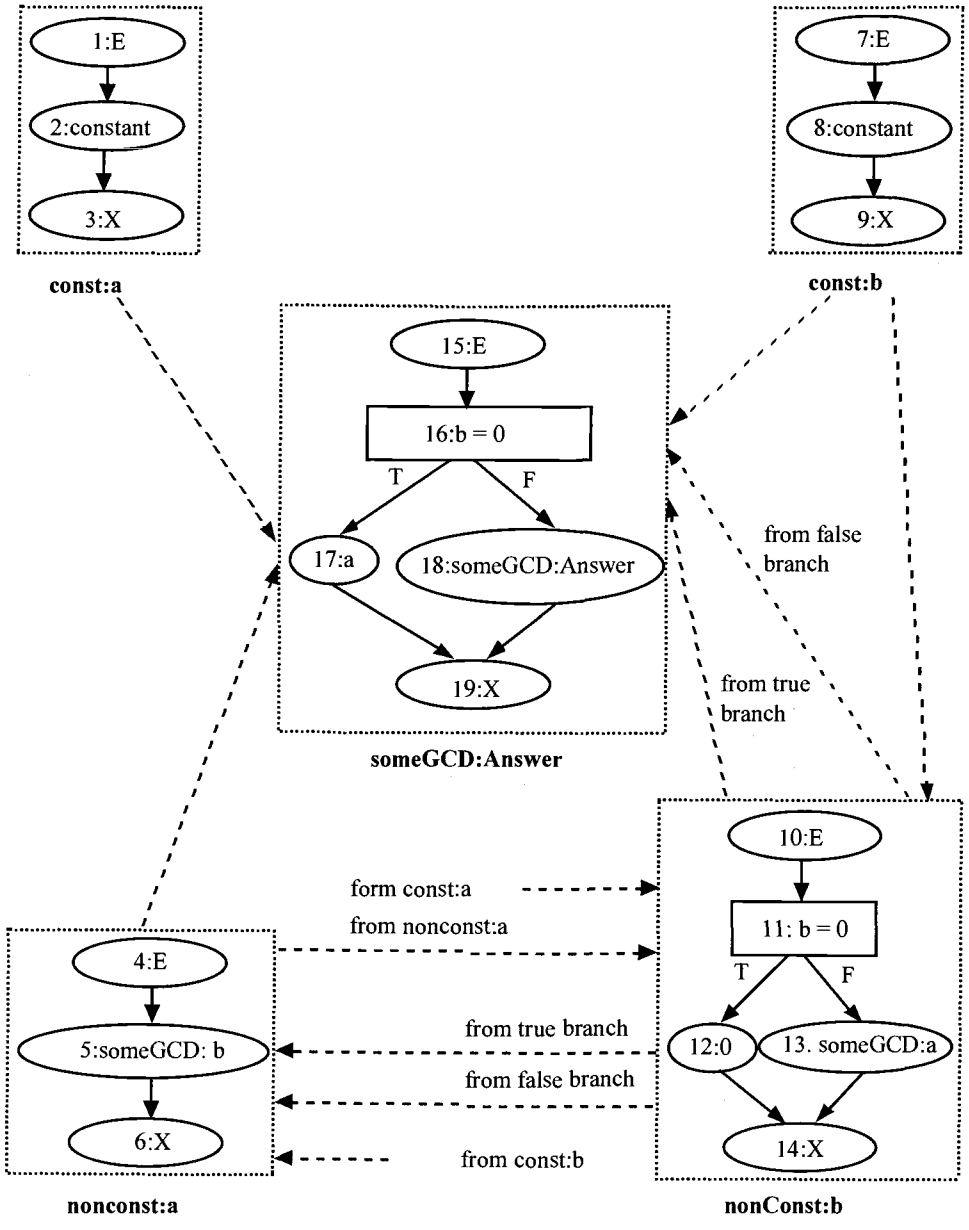


(Top)



(Bottom)

Figure 23. (Top): a GCD recursive program under the Model-only approach; (Bottom): the same program under the Copy Representative approach. Under the Model-only approach, the users can not validate the `Answer` cell in the copy spreadsheets, which prevents the user from being able to reach the same testedness as that under the Copy Representative approach.



Infeasible du-associations under both approaches:

(12,16 F).

Additional infeasible du-associations introduced by the Model-only approach:

(12, 16 T) and (8, 11T)

Figure 24. Partial CRG of GCD program under the Copy Representative approach.

5.4 Empirical Study[5]

5.4.1 Research Questions

To guide our choice of method, we conducted an experiment to compare the two approaches. Unfortunately, while testing the first group, the experimenters discovered an implementation bug that resulted in the loss of the group's data. Thus, the following results only compare the Copy-Representative group and the Extended WYSIWYT group. The specific objectives of the study were to investigate the following research questions:

Research Question 1. In which approach are spreadsheet programmers more effective in terms of du-adequacy?

Research Question2: In which approach do spreadsheet programmers have fewer redundant test cases?

Research Question 3: In which approach are spreadsheet programmers more effective at finding faults?

Research Question4: Which approach do spreadsheet programmers expect while testing?

5.4.2 Method and Procedures

The participants were 47 undergraduate and graduate computer science students enrolled in software engineering courses. Half of the subjects used the Copy Representative approach and half used the Extended WYSIWYT approach. Each of the students was given extra credit in their class for participation.

Subjects completed a background questionnaire and were given a 20-minute tutorial on Forms/3 that included a 2-minute open-ended practice session. Then, subjects performed three 5-minute testing sessions in which problem order was counter-balanced. Following the three sessions, participants completed a comprehension quiz intended to extract subjects'

[5] This study was done collaboratively with Andrew Ko.

understanding of the importance of testing, of how to choose appropriate test cases, and of the behavior of the underlying testing approach.

The subjects were given three recursive programs to find faults in: one calculated x^n, another calculated the greatest common divisor of two numbers, and the third calculated a class grade by accumulating scores from three copied spreadsheets. Before running the actual experiment, we evaluated its design, including the problems, tutorial, and user interface, using Cognitive Walkthroughs [Green et al. 2000] and pilot studies.

5.4.3 Results

Brief summaries of the analyses of the data are provided below. For all analyses, α = .05.

5.4.3.1 RQ1: Testing Coverage

To address RQ1 (testing coverage), the total spreadsheet testedness (du-associations covered out of total number of du-associations) for each problem was recorded for each subject (Table 11). A Repeated Measures ANOVA showed that the level of coverage of the Copy Representative group was significantly higher than that of the Extended WYSIWYT group across the three problems ($F=59.1$, $df=1,45$, $p<.001$). There was also a significant difference in coverage for the three problems ($F=9.0$, $df=2,44$, $p<.001$) and an interaction effect ($F=12.1$, $df=2,44$, $p<.001$), which says that the influence of the approach differed across problems.

	Subjects	Mean of coverage	Standard deviation of coverage
Copy Representative	24	0.671	0.085
Extended WYSIWYT	23	0.436	0.083

Table 11. Testing coverage statistics data. This data includes all three problems.

5.4.3.2 RQ2: Redundancy

To address RQ2 (redundancy), for each subject we recorded the percentage of redundant test cases out of the total number of recorded test cases (Table 12). A Repeated Measures ANOVA showed that the redundancy of the Copy Representative group was significantly greater than the redundancy of the Extended WYSIWYT group ($F=19.79$, $df=1,45$, $p<.001$). There were no differences in redundancy among the problems or interaction effects between the groups and the problems.

	Subjects	Mean of number of redundant test cases	Standard deviation of number of redundant test cases
Copy Representative	24	17	13.75
Extended WYSIWYT	23	12	10.20

Table 12. Redundancy statistics data. This data includes all three problems.

5.4.3.3 RQ3: Faults

To address RQ3 (faults), the number of faults found by each group was counted. Though the two groups did not differ significantly in the number of faults they found, more subjects from the Copy Representative group found all faults and fewer found no faults than the Extended WYSIWYT group (Table 13).

	No bugs	1 bug	2 bugs
Copy Representative	2	7	15
Extended WYSIWYT	6	7	10

Table 13. The number of faults found in each group.

5.4.3.4 RQ4: Expectation

To gather data about the groups' understanding of their respective testing approach (RQ4), we asked subjects which of four cells on model and copy spreadsheets would become more tested if a cell on a copied spreadsheet was validated in an example problem. In both groups, approximately 17% answered all questions correctly for the given approach (Table 14).

The same data were also analyzed regarding whether the subjects expected testedness information to be passed onto the model spreadsheet when they validated a cell on a copied spreadsheet. A binomial test for proportions revealed that more subjects expected the model spreadsheet to share testing information with its copies ($p<.01$). The magnitude of this expectation was not different between groups ($\chi^2=.537$, $p>.1$).

	Subjects	Mean of scores	Standard deviation of scores
Copy Representative	24	0.375	0.77
Extended WYSIWYT	23	-0.130	0.97

Table 14. Expectation statistics data. Scores were –1.0 or 1.0, indicating inconsistency or consistency respectively with the approach the subject used.

5.4.3.5 Other Results

A Repeated Measures ANOVA was also performed on the total number of test cases each subject performed (Table 15), and it was found that the Extended WYSIWYT group performed significantly more test cases than did the Copy Representative group ($F = 8.52$, $df = 1, 45$, $p < .001$). There was no significant difference in the number of test cases between problems, but there was a significant interaction between the testing approach and the problems ($F = 3.68$, $df = 2, 44$, $p < .05$).

	Subjects	Mean of number of test cases	Standard deviation of number of test cases
Copy Representative	24	12	3.81
Extended WYSIWYT	23	28	6.68

Table 15. Test cases performed. This data includes all three problems.

5.4.4 Discussions

Two issues regarding testing are whether users can achieve more coverage with less work and whether increasing amount of work by users leads to finding more of the faults. The Copy Representative group achieved much higher testing coverage with fewer clicks, while the Extended WYSIWYT group worked harder but achieved less coverage. With the Extended WYSIWYT approach, users are forced to test each spreadsheet independently, and thus it requires more work to achieve the same level of coverage. One might hypothesize that the Extended WYSIWYT group would find more faults since they did more tests. However, the results of the experiment show that the groups did not differ in their ability to find faults, and in fact, the Copy Representative group found a few more overall.

The results of the experiment also showed that the Copy Representative group performed more redundant test cases while achieving higher coverage. Redundancy is a two-sided issue: on the one hand, Copy Representative users can be viewed as "wasting effort;" on the other hand, the Extended WYSIWYT group could be viewed as "wasting effort" in that they tested the same formulas on multiple spreadsheets. Regarding understandability, although results indicated that most subjects could not accurately predict the behavior of either of the testing approaches, significantly more subjects expected the behavior to be that of the Copy Representative approach.

In summary, the experiment certainly did not reveal the understandability advantage of the Extended WYSIWYT approach that we initially expected. Also, from a theoretical standpoint, the Copy Representative approach is better because it avoids the thorny theoretical problems raised by the Extended WYSIWYT approach. Taken together, these two factors suggest that the Copy Representative approach is the better choice.

Chapter 6: Testing User-Defined Abstract Data Types

As described in Section 2.6, Forms/3 supports user-defined abstract data types. A special cell – *abstraction box* (abbreviated as *absBox*) – defines the composition of the type, and its value is an instance of the type. An absBox can contain many interior cells as Figure 25 shows. The implications of this are that an absBox's default (implicit) formula is the composition of its interior cells values, and that the interior cells' default (implicit) formulas are the decomposition of the absBox's value. However, this advanced programming feature of Forms/3 had not been supported by the WYSIWYT methodology. In this chapter, we extend the WYSIWYT methodology to support user-defined abstract data types, specifically supporting absBox cells and interior cells of absBox cells.

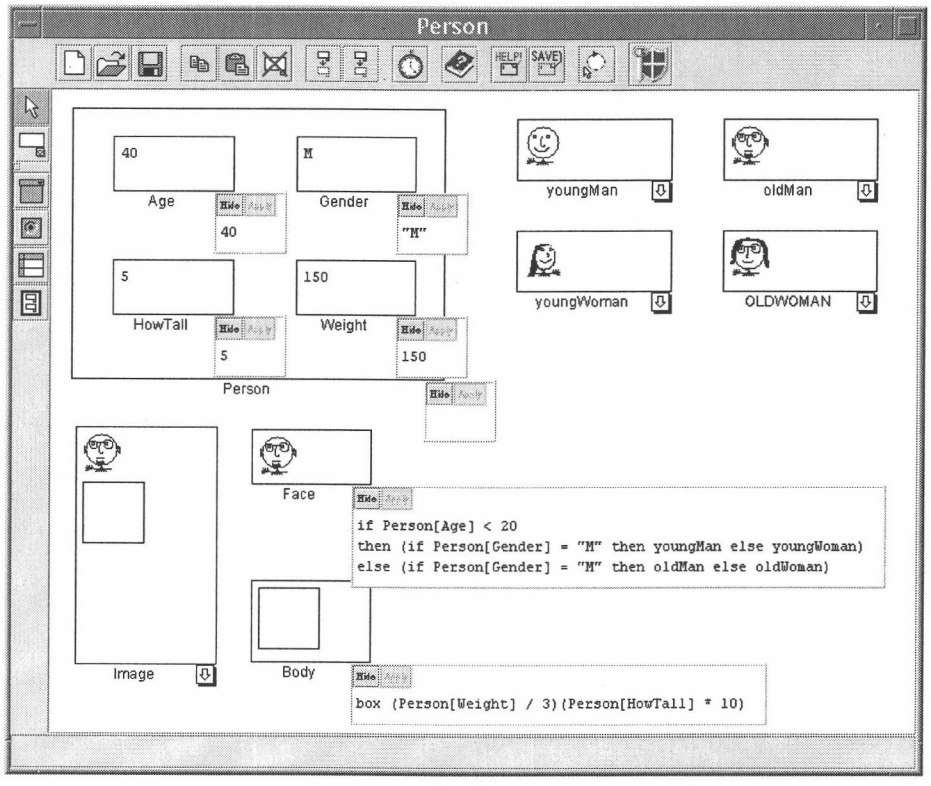


Figure 25. A use-defined abstract data type – `Person`. `Person` is a user-defined abstract data type which is composed of 4 attribute cells – `Age`, `Gender`, `HowTall` and `Weight`.

6.1 Design View of testing user-defined abstract data types

As mentioned in the previous chapters, the underlying assumption of the WYSIWYT methodology is that as the user develops a spreadsheet incrementally, he or she could also be testing incrementally. We would like to have our testing algorithms for absBox cells be consistent with this assumption. To allow this, we must incrementally maintain WYSIWYT data in each of the following cases:

Case 1: Neither an absBox nor its interior cells have formulas. For example, prior to the state shown in Figure 25, several cells such as *Age*, *Gender*, *HowTall* and *Weight* were added to the absBox cell *Person* initially without entering their formulas.

Case 2: An absBox cell has a formula and none of its interior cells have formulas. For example, in Figure 26, the user edits the absBox cell *Person*'s formula and makes it a reference to *Person1*, by which, all interior cells will dynamically attain values from its absBox cell.

Case 3: Each interior cell in an absBox cell has a formula and the absBox does not have a formula. For example, in Figure 25, the user edits the formula of each interior cell and gets an instance of the type *Person* of which the *Person*'s Age is 30, Gender is "M", HowTall is 6 feet and Weight is 150 pounds.

Case 4: An absBox has a formula and some of its interior cells have formulas. For example, in Figure 28, the user first entered *Person*'s formula and then entered its interior cell *Age*'s formula.

Since no input value is available, in Case 1 there is no need for testing.

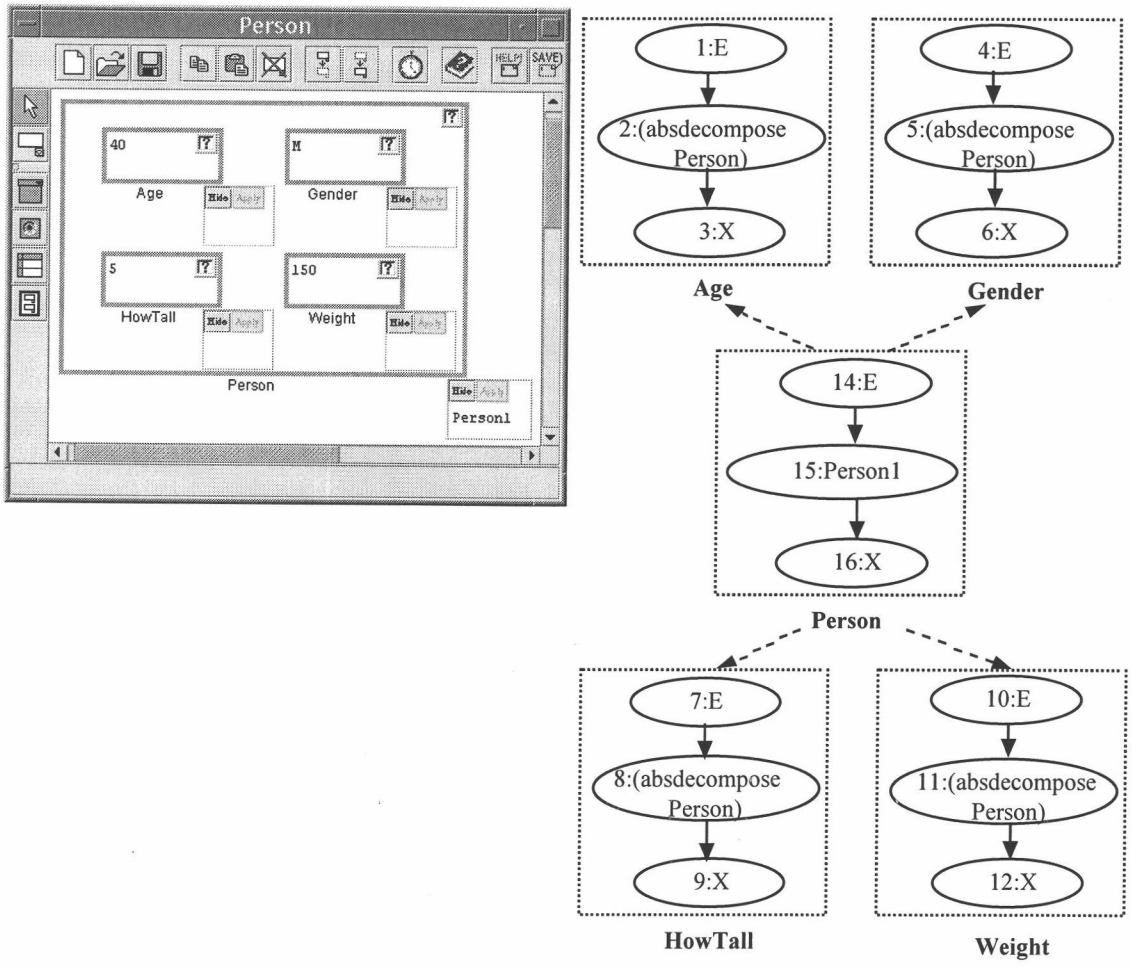


Figure 26. Case 2 and partial CRG in Case 2. (Left) Case 2: the absBox cell `Person` having a formula and all its interior cells having no formulas and attaining their values from Person; (Right) partial cell relation graph in Case 2.

6.1.1 Testing in Case 2

In Case 2, each interior cell attains its value from its absBox cell. Thus the implicit formula for each interior cell is that it is a component of the decomposed `Person` defined by the absBox. The cell relation graph (CRG) for the user-defined abstract data type based on this dataflow relationship is shown in Figure 26.

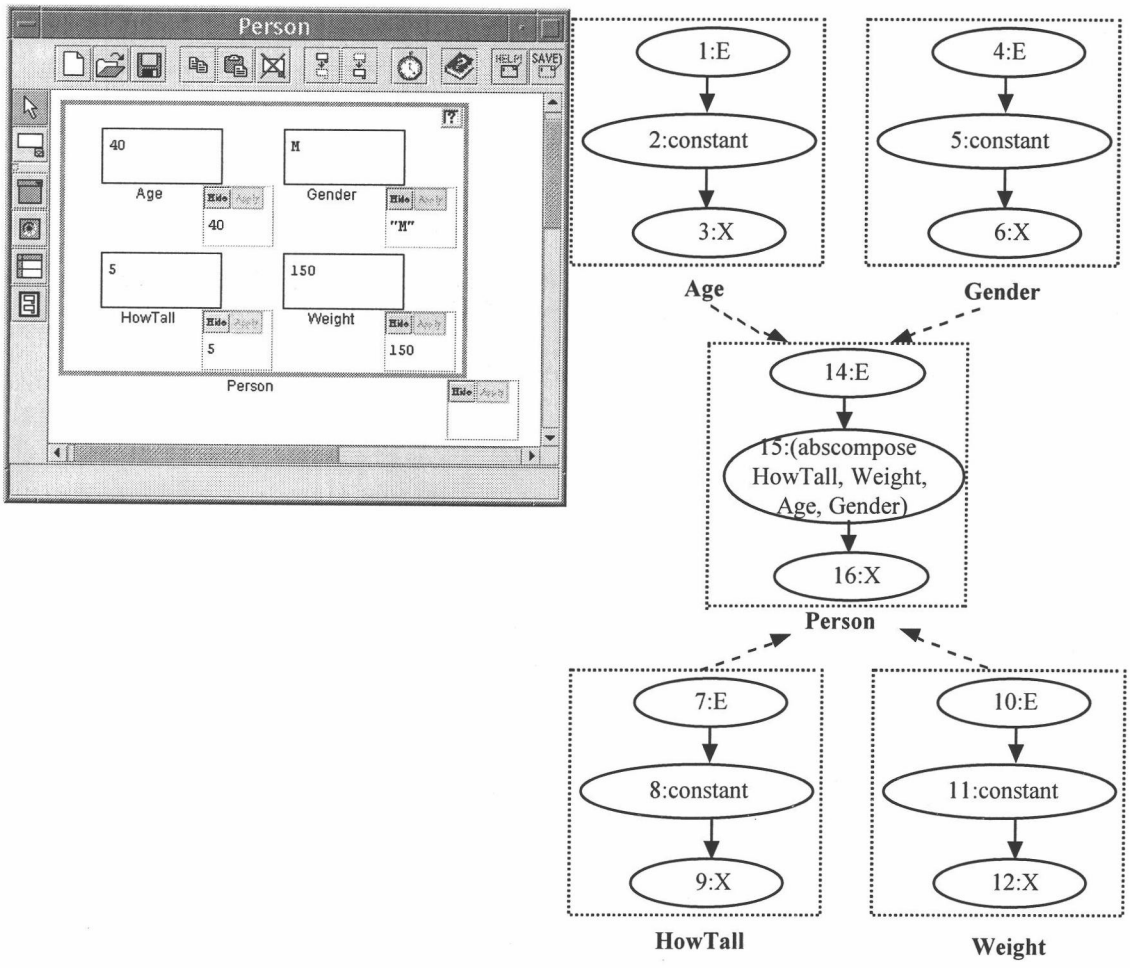


Figure 27. Case 3 and partial CRG in Case 3. (Left) Case 3: the absBox cell `Person` having no formula and all its interior cells having formulas and `Person` attaining its values from composition of its interior cells; (Right) partial cell relation graph in Case 3.

6.1.2 Testing in Case 3

In Case 3, each interior cell has a formula and the absBox cell's value is simply a composition of all its interior cells. For example, as in Figure 27, there is no explicit formula for the absBox cell, and the absBox's value is simply a composition of all its interior cells. For this kind of dataflow relationship, the cell relation graph will be as in Figure 27, in which all interior cells are definitions and the absBox cell is the use.

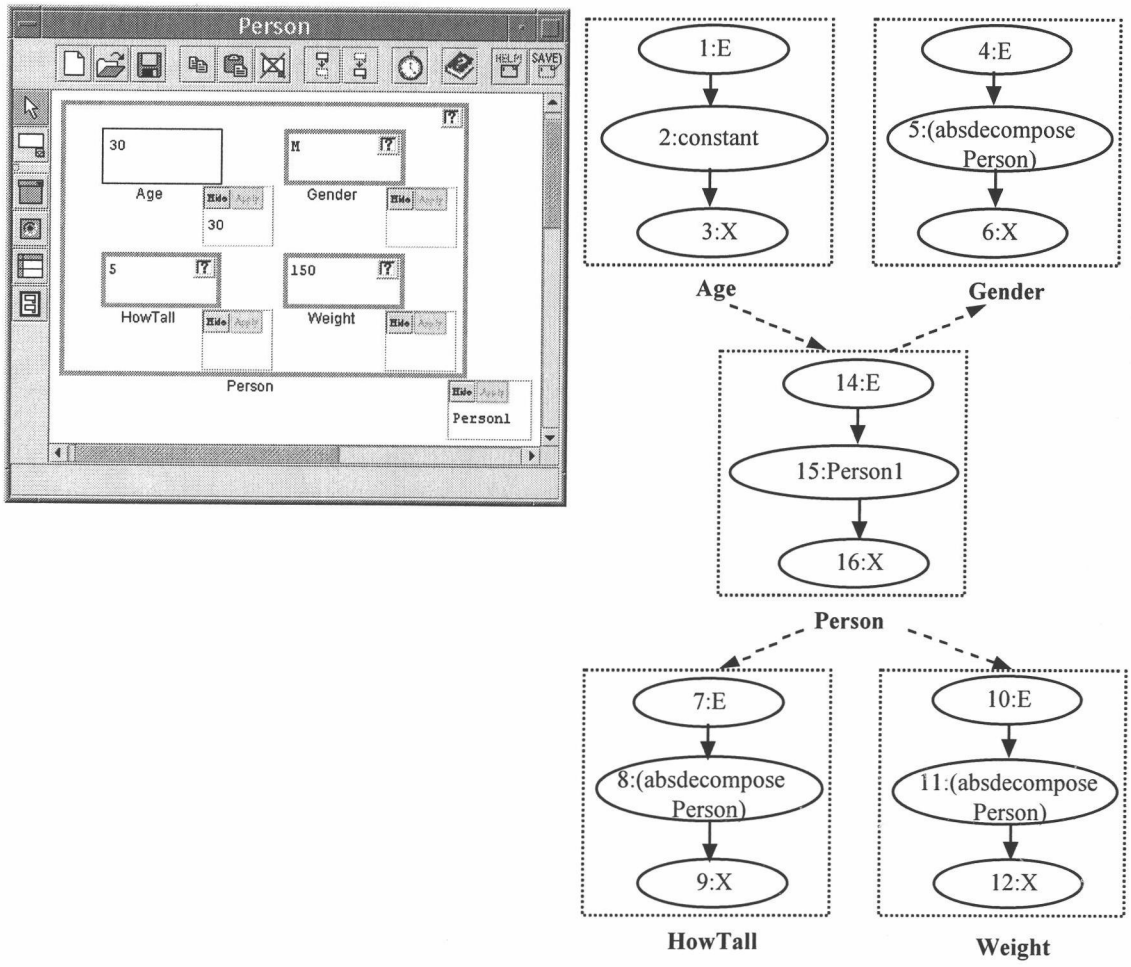


Figure 28. Case 4 and partial CRG in Case 4. (Left) Case 4: the absBox cell `Person` having a formula and one of its interior cells – `Age` – having a formula; (Right) partial cell relation graph in Case 4.

6.1.3 Testing in Case 4

If there is an explicit formula for the absBox cell and some interior cells also have formulas (Case 4), we would have a cell relation graph such as shown in Figure 28 shows. In Figure 28, the cell `Age` has an explicit formula that has an effect on the value of the absBox cell `Person`; whereas, the rest of the interior cells do not have explicit formulas and they attain their values from decomposing `Person`'s value; hence, other cells are uses of the cell `Person`.

6.2 Special considerations before implementation

The characteristics of the cases in user-defined abstract data types raise some considerations described as follows.

6.2.1 How to trigger Task 1

In the original WYSIWYT methodology, Task 1 is triggered when a cell's formula is entered. In the process of Task 1, a cell `C`'s incoming du-associations as well as outgoing du-associations are collected. We used the algorithm described in Section 5.3.4, in which the cell `C`'s incoming du-associations are collected first and then `C`'s outgoing du-associations are collected through using existing – old du-associations – information. The reason this method for collecting `C`'s outgoing du-associations works is that there exist outgoing du-associations information for `C`, which were previously collected for `C`'s direct consumers when they were entered.

To collect du-associations for a user-defined abstract data type, we cannot use that algorithm any more, since the assumption of pre-collected outgoing du-associations is not valid. Instead of only collecting incoming du-associations initially, the system must collect outgoing du-associations between the absBox cell and its interior cells at the execution time as well. In Case 2, for example, in Figure 26, if the user enters the absBox cell `Person`'s formula and makes a reference to another absBox cell `Person1`, the `Person` is the definition implicitly referenced by all its interior cells as described in Section 6.1.1. At this moment, the system is supposed to collect outgoing du-associations from `Person` to each of its interior cells. However, if using the original algorithm, no such outgoing du-associations could be collected since no pre-collected du-associations are available. In Case 3 and Case 4, for example, in Figure 27 and Figure 28, if the user enters a formula for the interior cell `Age`, the `Age` is the definition implicitly referenced by its absBox cell as described in Section 6.1.2 and Section 6.1.3. At this moment, the system should collect outgoing du-associations from `Age` to `Person`. But in the original algorithm, it is impossible because there are no pre-collected du-associations available.

6.2.2 Implicit vs. explicit

In the original WYSIWYT methodology, when we talked about a CRG, each node V – a formula graph - was based on an explicit formula of a cell. A du-association is a pair between a definition in a cell D's explicit formula and an explicit reference to D in a cell C's formula. They are *explicit definitions and uses*. However, in user-defined abstract data types, there are also implicit formulas and therefore *implicit definitions and uses*. For example, in Figure 27, Person has an implicit formula - (abscompose) – node 15 is an implicit definition and also implicit uses of nodes 2, 5, 8, 11. We term this kind of definition an *implicit definition*, the use an *implicit use,* and the du-association between an (implicit or explicit) definition of a cell and an implicit use of another cell an *implicit du-association*.

6.3 Algorithms

To support testing user-defined abstract data types, the system will perform the four tasks described in Section 2.3. Task 2, Task 3 and Task 4 need not be changed. However, Task 1 needs to be modified to reflect the characteristics of user-defined abstract data types described in the last section.

We introduced a special type of use and du-association – *implicit use* and *implicit du-association* - in our implementation. Both are inherited classes from the class use and the class du-association with a field saying they are implicit. We didn't introduce an implicit definition because the system actually already kept such definitions. For example, if a cell C has no explicit formula and is referenced by some cell D, the system will collect incoming du-associations as if C had an explicit formula.

At the top of Figure 29, the algorithm *CollectAssoc* is a modified version of the algorithm described in Section 5.3.3.

If in Case 2, cell C is an absBox cell with an explicit formula and has not collected implicit outgoing du-associations yet, the system will call *CollectImplicitOutgoingAssoc* to collect outgoing implicit du-associations from C to C's interior cells and then delete all old implicit incoming du-associations of C if applicable.

```
Algorithm CollectAssoc (C, SharedGraph)
Let oldSharedGraph = C.SharedGraph
// (1) collecting incoming information
For each use ∈ SharedGraph.Uses
  Let useCells = {copies sharing SharedGraph}
  Let allDefCells = { definition cells that contribute to useCells }
  Let defSharedGraphs = {defCell.SharedGraph | defCell ∈ allDefCells}
  For each defSharedGraph ∈ defSharedGraphs
    For each def ∈ defSharedGraph.Defs
      Let DUA = (def, use, false)
      Add DUA to SharedGraph.DUAs.Incoming
      Add DUA to defSharedGraph.DUAs.Outgoing
// (2) collecting outgoing information
For def ∈ SharedGraph.Defs
For each DUA' ∈ oldSharedGraph.DUAs.outgoing
      Let use = DUA'.Use
      Let useSharedGraph be the SharedGraph containing use
      Let DUA = (def, use, false)
      Add DUA to SharedGraph.DUAs.Outgoing
      Add DUA to useSharedGraph.DUAs.Incoming
If C is an absBox cell without implicit outgoing du-associations
   or C is an interior cell with an explicit formula and C has no
      implicit outgoing du-associations
  CollectImplicitOutgoingAssoc ( C )
  Let incomingImplicitDUAs = { aDUA is an implicit du-association
      | aDUA ∈ C.SharedGraph.DUAs.Incoming }
  DeleteDUAs (incomingImplicitDUAs)
If C is an interior cell without an explicit formula and C's absBox
      cell has an explicit formula
  CollectImplicitIncomingDUAs ( C )
DeleteSharedGraphInformation(oldSharedGraph)

Algorithm CollectImplicitOutgoingAssoc ( C )
  If C is an absBox cell
    Let childrenCells be all interior cells of C
    For each child ∈ childrenCells
      Initialize an implicit-use and add it to child.SharedGraph.Uses
      For each def ∈ C.SharedGraph.Defs
        Let implicit-DUA = (def, implicit-use, false)
        Add implicit-DUA to C.SharedGraph.DUAs.outgoing
        Add implicit-DUA to child.SharedGraph.DUAs.incoming
  If C is an interior cell of absBox cell
    Let parent be C's absBox cell
    Initialize an implicit-use and add it to parent.SharedGraph.Uses
    For each def ∈ C.SharedGraph.Defs
      Let implicit-DUA = (def, implicit-use, false)
      Add implicit-DUA to C.SharedGraph.DUAs.outgoing
      Add implicit-DUA to parent.SharedGraph.DUAs.incoming

Algorithm CollectImplicitIncomingAssoc ( C )
  Let parent be C's absBox cell
  Initialize an implicit-use and add it to C.SharedGraph.Uses
  For each def ∈ parent.SharedGraph.Defs
    Let implicit-DUA = (def, implicit-use, false)
    Add implicit-DUA to C.SharedGraph.DUAs.incoming
    Add implicit-DUA to parent.SharedGraph.DUAs.outgoing
```

Figure 29. Algorithms of collecting implicit du-associations for testing the user-defined abstract data types. The gray part of the algorithm of *CollectAssoc* is identical to that in Figure 20.

If in Case 2, cell *c* is an interior cell without an explicit formula that is added to its absBox cell, the system will call *CollectImplicitIncomingAssoc* to collect incoming du-associations from the absBox cell to *c*.

If in Case 3 or Case 4, cell *c* is an interior cell with an explicit formula that has not collected implicit outgoing du-associations yet, the system will call *CollectImplicitOutgoingAssoc* to collect outgoing implicit du-associations to *c*'s absBox cell and then delete all old implicit incoming du-associations of *c* if applicable.

6.4 Time complexities

The algorithm *CollectImplicitOutgoingAssoc* and *CollectImplicitIncomingAssoc* are called by the algorithm *CollectAssoc*. The algorithm *CollectAssoc* has a time complexity of $O(p_d + c_d)$ described in Section 5.2.4.1. In the worst case – when *c* is an absBox cell with an explicit formula – the system calls the algorithm *CollectImplicitOutgoingAssoc*, which has two loops, one nested within the other, that add to the cost of *CollectAssoc*. Let c_d' be the number of interior cells of the absBox cell *c*; let h be the number of definitions that *c* can have. The total time cost of the algorithm *CollectImplicitOutgoingAssoc* is:

$O(c_d'h)$

The above cost can be further simplified because, as pointed out before, most spreadsheet languages have a maximum on formula length. Hence, h becomes constant-bounded by the maximum formula length. Thus, the simplified asymptotic time cost of *CollectImplicitOutgoingAssoc* is:

$O(c_d')$

In addition, $c_d' < c_d$ (c_d is number of direct consumers of *c*) due to the fact that all interior cells in an absBox cell are direct consumers of the absBox cell if the absBox cell has an explicit formula and interior cells attain value from the absBox cell.

The algorithm *CollectImplicitIncomingAssoc* only has one loop which is the number of definitions that the absBox cell have - constant-bounded - thus it has O(1) time complexity.

Overall, the algorithms for testing user-defined abstract data types add no new multiplicative factors to the existing testing subsystem and the system keeps Task 1's time complexity as: $O(p_d + c_d)$. Because of this, the cost of Task 1 in context remains of the same order as the system evaluation and notification tasks as described in Section 5.3.4.2.

Chapter 7: Conclusion

In this thesis, we have presented extensions of the original WYSIWYT methodology to some advanced features of spreadsheet languages. This includes improvements in visual devices for testing grids, support for testing recursive programs, and support for testing user-defined abstract data types.

Prior to this work, researchers in our group have partially developed two approaches to testing large grids in spreadsheets [Sheretov 2000, Burnett et al. 1999], namely the Straightforward approach and the Region Representative approach. Both approaches allow one user validation action to be leveraged across the entire region, which reduces user actions. Because the Region Representative approach allows more information sharing among cells with the same formulas, for behind-the-scenes reasoning, this approach is more scalable to very large grids. In this thesis, by improving visual devices through sharing testing bounds and data flow arrows among cells with the same formulas under the Region Representative approach, the GUI updating was made more consistent with the sharing principle behind-the-scenes. Also, by performing performance experiments, we have experimental information about how the scalability of Region Representative approach compares to that of the Straightforward approach. The performance comparisons showed that the Region Representative's time savings increased as the size of the spreadsheet grew.

In this document, we also presented two new approaches to testing recursive spreadsheets. The approaches presented both extend the basic WYSIWYT approach to support recursion. The Extended WYSIWYT approach is dataflow-based, as is the original WYSIWYT methodology, but it raises several testing-theoretic issues. However, its consistency with basic WYSIWYT would seem to make it most useful to the humans actually using it. The Copy Representative approach honors not only dataflow dependencies, but also shares testedness information among multiple copies of the same cell. This allows the user to avoid duplicating testing of identical logic and also avoids the theoretical problems raised in the Extended WYSIWYT approach.

To help inform our choice between these two approaches, we implemented both and conducted an empirical study. Users of the Copy Representative approach achieved more testing coverage. Their efforts to achieve this included more redundant testing, which can be

viewed as either a greater "safety net" or wasted effort. Neither of the groups predicted behavior accurately - only 17% subjects in both groups answered all questions correctly - but their expectations of propagation of testedness were that of the Copy Representative approach. These results, combined with its theoretical advantages, lead us to view the Copy Representative approach as the best choice for supporting testing of recursive programs in this kind of language.

To support testing user-defined abstract data types, we introduced implicit uses and implicit du-associations in our implementation. An absBox or an interior cell in an absBox may or may not have an explicit formula. If an interior cell has an explicit formula, it becomes a definition cell and its enclosing absBox becomes an implicit use of it, because basically an absBox's value is simply a composition of all its interior cells. If an absBox has a formula, it becomes all its interior cells' definition and the interior cells have an implicit use of it, because the interior cells' value comes from decomposing the absBox's formula.

Some future work may include empirical comparison between the Region Representative approach and the Straightforward approach, performance experiments on the two approaches to test recursive programs, and investigation of ease of use for the absBox testing approach.

Bibiliography

[Aho et al. 1986] A. Aho, R. Sethi, and J. Ullman, "Compilers, Principles, Techniques, and Tools," *Reading, MA: Addison-Wesley Publishing Company*, 1986.

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

[Ambler 1999] A. Ambler, "The Formulate visual programming language," *Dr. Dobb's Journal*, 21-28, Aug. 1999.

[Burnett and Gottfried 1998] M. Burnett and H. Gottfried, "Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures," *ACM Transactions on Computer-Human Interaction* 5(1), 1-33, Mar. 1998.

[Burnett et al. 1999] M. Burnett, A. Sheretov, and G. Rothermel, "Scaling Up a 'What You See Is What You Test' Methodology to Testing Spreadsheet Grids," *Proceedings of 1999 IEEE Symposium on Visual Languages*, 30-37, Sept. 1999.

[Burnett et al. 2001] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang, "Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm," *Journal of Functional Programming*, accepted (to appear).

[Chi et al. 1998] E. Chi, J. Riedl, P. Barry, and J. Konstan, "Principles for Information Visualization Spreadsheets," *IEEE Computer Graphics and Applications*, 30-38, July/Aug. 1998.

[Duesterwald et al. 1992] E. Duesterwald, R. Gupta, and M. L. Soffa, "Rigorous Data Flow Testing through Output Influences," *Proceedings of Second Irvine Software Symposium*, 131-145, Mar. 1992.

[Frankl and Weyuker 1998] P. Frankl and E. Weyuker, "An Applicable Family of Data Flow Criteria," *IEEE Transactions on Software Engineering* 14(10), 1483-1498, Oct. 1988.

[Green et al. 2000] T. Green, M. Burnett, A. Ko, K. Rothermel, C. Cook, J. Schonfeld, "Using the cognitive walkthrough to improve the design of a visual programming experiment," *IEEE Symposium on Visual Languages*, Seattle, WA, 172-179, Sept. 2000.

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

[Harrold and Soffa 1988] M. J. Harrold and M. L. Soffa, "An Incremental Approach to Unit Testing during Maintenance," *Proceedings of Conference on Software Maintenance*, 362-367, Oct. 1988.

[Horgan and London 1991] J. Horgan and S. London, "Data Flow Coverage and the C Language," *Proceedings of the Fourth Symposium on Testing, Analysis, and Verification*, 87-97, Oct. 1991.

[Krishna et al. 2001] V. Krishna, C. Cook, D. Keller, J. Cantrell, C. Wallace, M. Burnett, G. Rothermel, "Incorporating incremental validation and impact analysis into spreadsheet maintenance: An empirical study", *TR 01-60-06, Oregon State University*, Jan. 2001.

[Laski and Korel 1993] J. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Transactions on Software Engineering* 9, 347-354, May 1993.

[Leopold and Ambler 1997] J. Leopold and A. Ambler, "Keyboardless Visual Programming Using Voice, Handwriting, and Gesture," *Proceedings of 1997 IEEE Symposium on Visual Languages*, 28-35, Sept. 1997.

[Myers 1991] B. Myers, "Graphical Techniques in a Spreadsheet for Specifying User Interfaces," *Proceedings of ACM Conference on Human Factors in Computing Systems*, 243-249, May 1991.

[Myers et al. 1990] B. Myers, D. Guise, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal, "Garnet: Comprehensive support for graphical, highly interactive user interfaces," *Computer*, 71-85, Nov. 1990.

[Ntafos 1984] S. Ntafos, "On Required Element Testing," *IEEE Transactions on Software Engineering* SE-10, 795-803, Nov. 1984.

[Offutt et al. 1996] J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," *Software Practice and Experience* 26, 165-176, Feb. 1996.

[Pande et al. 1994] H. D. Pande, B. G. Ryder and W. Landi, "Interprocedural Def-Use Associations in C programs," *IEEE Transactions on Software Engineering* 20(5), 385-403, May 1994.

[Panko 1998] R. Panko, "What we know about spreadsheet errors," *Journal of. End User Computing*, 15-21, Spring 1998. (Also available at: http://panko.cba.hawaii.edu/ssr/).

[Panko and Halverson 1996] R. Panko and R. Halverson, "Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks," *Proceedings of Hawaii International Conference on System Sciences*, Jan 1996.

[Perry and Kaiser 1990] D. Perry and G. Kaiser, "Adequate Testing and Object-oriented Programming," *Journal of Object-Oriented Programming* 2, 13-19, Jan. 1990.

[Rapps and Weyuker 1985] S. Rapps, and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering* 11, 367-375, Apr. 1985.

[Reichwein et al. 1999] J. Reichwein, G. Rothermel, and M. Burnett, "Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging," *Proceedings of Conference on Domain Specific Languages (DSL'99)*, 25-38, Oct. 1999.

[Rothermel and Harrold 1997] G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Transactions on Software Engineering and Methodology* 6, 173-210, Apr. 1997.

[Rothermel et al. 1997] G. Rothermel, L. Li, and M. Burnett, "Testing Strategies for Form-based Visual Programs," *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 96-107, Nov. 1997.

[Rothermel et al. 1998] G. Rothermel, L. Li, C. DuPuis, and M. Burnett, "What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs," *Proceedings of International Conference on Software Engineering*, 198-207, Apr. 1998.

[Rothermel et al. 2001] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov, "A Methodology for Testing Spreadsheets," *ACM Transactions on Software Engineering and Methodology*, Jan. 2001.

[Rothermel et al. 2000] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, and G. Rothermel, "An Empirical Evaluation of a Methodology for Testing Spreadsheets," *Proceedings of International Conference on Software Engineering*, 230-239, June 2000.

[Sheretov 2000] A. Sheretov "A Methodology for Testing Spreadsheet Grids," *Master thesis, Oregon State University*, Jan. 2000

[Smedley et al. 1996] T. Smedley, P. Cox, and S. Byrne, "Expanding the Utility of Spreadsheets Through the Integration of Visual Programming and User Interface Objects," *ACM Proceedings of Workshop on Advanced Visual Interfaces*, Gubbio, Italy, 148-155, May 27-29, 1996.

[Wang and Ambler 1996] G. Wang and A. Ambler, "Solving Display-Based Problems," Proceedings of 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 122-129, Sept. 3-6, 1996.

[Weyuker 1986] E. Weyuker, "Axiomatizing Software Test Data Adequacy," *IEEE Transactions on Software Engineering* 12, 1128-1138, Dec. 1986.

[Weyuker 1993] E. Weyuker, "More Experience with Dataflow Testing," *IEEE Transactions on Software Engineering* 19(9), 912-919, Sept. 1993.

[Wilde and Lewis 1990] N. Wilde and C. Lewis, "Spreadsheet-Based Interactive Graphics: From Prototype to Tool," *Proceedings of ACM Conference on Human Factors in Computing Systems*, 153-159, Apr. 1990.

[Yang and Burnett 1994] S. Yang and M. Burnett, "From concrete forms to generalized abstractions through perspective-oriented analysis of logical relationships," *IEEE Symposium on Visual Languages*, St. Louis, MO, 6-14, Oct. 1994.