Visually Testing Recursive Programs in Spreadsheet Languages[*]

Margaret Burnett, Bing Ren, Amy J. Ko, Curtis Cook, Gregg Rothermel
Department of Computer Science, Oregon State University, Corvallis, OR 97331
{burnett, ren, koan, cook, grother}@cs.orst.edu

Abstract

Although there has been recent research into ways to design visual programming languages and environments, little attention has been given to systematic testing in these languages, and what work has been done does not address "power" features such as recursion. In this paper, we discuss two possible ways the "What You See Is What You Test" methodology could be extended to accommodate recursion. The approaches are presented in terms of their testing theoretic aspects and then implementation strategies and algorithms. Since the ultimate goal is to help the people using these languages, we also present an empirical study and use its results to inform our choice as to which of the two approaches to adopt.

1. Introduction

Although there has been a fair amount of research into mechanisms for coding and understanding programs written in visual programming languages, there has been little attention to other aspects of the software lifecycle for programs written in these languages. To address one such aspect, we have been working on a visual testing methodology for testing programs written in declarative visual languages, prototyping our results in spreadsheet languages—the most widely used type of visual programming language in practice. We use the term "spreadsheet language" to denote not only commercial spreadsheet systems, but also research spreadsheet languages for a wide variety of purposes, such as for producing high-quality visualizations of complex data [5], for specifying GUI interfaces [9, 16], for manipulating matrices [1], and for working with user-defined objects [2, 3]. Our visual testing methodology is called the "What You See Is What You Test (WYSIWYT)" methodology [13, 14].

The testing problem is particularly challenging for visual programming languages that aim toward diverse audiences ranging from end users to professional programmers, as is the case with spreadsheet languages. In such languages, the testing approach must make sense to end users, while at the same time supporting an organized approach to testing backed by a firm foundation to provide the testing power needed by professional programmers. Further, it is well documented that spreadsheets contain many faults (logic errors); virtually every study that looks for faults in spreadsheets finds them. (See [11] for a survey of this work.) Finally, spreadsheet languages' highly interactive, incremental characteristics such as automatic and immediate visual feedback impose the constraint that a testing methodology for such languages must also be interactive and incremental, and must feature automatic and immediate visual feedback as well. For these reasons, spreadsheet languages make a particularly good "acid test" for an approach to a testing methodology for declarative visual programming languages.

Our previous work presented the WYSIWYT methodology for individual spreadsheet cells [14], later extended it to large grids in which some cells share the same formula [4], and empirically validated its usefulness to both programmers and end users [6, 15]. 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 paper, we extend the WYSIWYT methodology to support recursive spreadsheets and linked copied spreadsheets in general.

2. Spreadsheet Design Patterns

Recursive programs in languages supporting end users can include programming structures that are rarely found in traditional languages, and this is demonstrated well by spreadsheets. Design patterns have become widely used in understanding program structures in traditional programming paradigms, and we will use that device here.

In Figure 1, four linked spreadsheet design patterns are shown. The dataflow arrows in the graphs represent cell references, and the nodes represent the spreadsheets.

[*] This work was supported in part by NSF under ITR-0082265, CCR-9806821, and CAREER Award CCR-9703198.

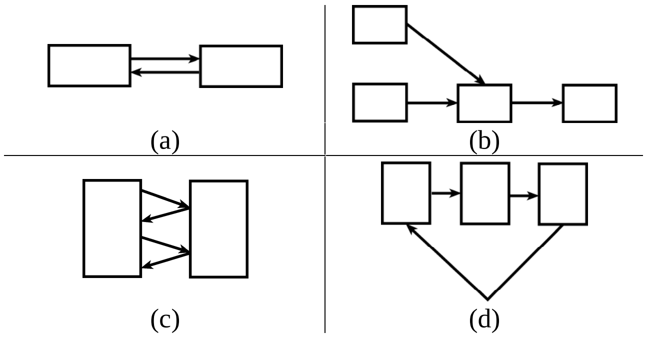


Figure 1: Four design patterns for linked spreadsheets. Arrows are in the direction of dataflow. (a) Analogous to call-return. (b) Pipeline. (c) Analogous to co-routines. (d) Pipeline with return-to-start.

Figure 1(a) shows the equivalent of a traditional call-return relationship of one procedure calling another; in a recursive situation, the procedure/spreadsheet on the left is a copy of the procedure/spreadsheet on the right. However, as Figure 1(b) shows, linked spreadsheets, some of which can be copies of the others, can exist without a complete call-return relationship, such as in a pipeline-like arrangement. In contrast to this, in traditional languages, pipelines are not usually allowed. Figure 1(c) shows what might be classified as "co-routines" in traditional programming literature, in which some cell on the left spreadsheet references a cell on the right, which in turn references another cell on the left, and so on. Figure 1(d) is a combination of (a) and (b).

As these examples show, spreadsheet languages allow non-traditional programming structures. This suggests that a testing methodology for such languages must be flexible enough to handle cell referencing and linked spreadsheet "design patterns" beyond the ubiquitous call-return pattern of traditional programs.

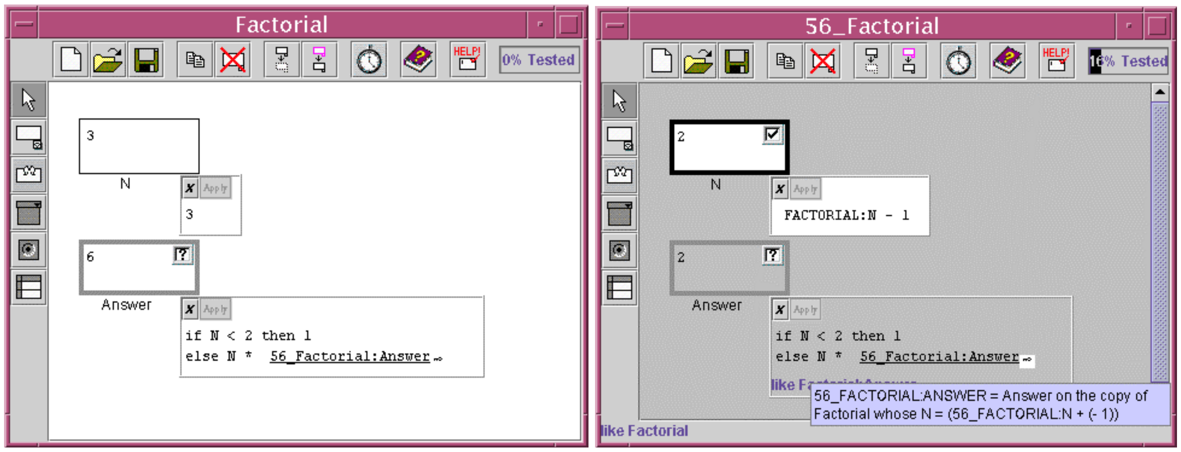
3. Testing Recursive Spreadsheets

Prior to this work, we developed WYSIWYT for spreadsheets without recursion and prototyped it in the visual spreadsheet language Forms/3 [2, 3]. With the WYSIWYT methodology, cells are initially colored with red borders (red means "untested"). If the user *validates* a cell's value by checking it off in a checkbox in the cell's corner, borders change color along a continuum of red to blue ("untested" to "tested").

3.1 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. Figure 2 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. (The notation S:X means a cell X on spreadsheet S. For example, Factorial:N means cell N on a spreadsheet named Factorial.) The main thing to notice in the figure at this point is how the WYSIWYT methodology appears to the user. Red-bordered cells (light gray) are untested, blue-bordered cells (black) are tested, and purples (grays) are between. A user can check off a correct value, as in 56_Factorial:N. If there is a question mark in a cell, checking off the cell will increase testedness. The upper right corner of each spreadsheet reports a spreadsheet's overall testedness.

To reason about testedness in the Extended WYSIWYT approach as well as in the original WYSIWYT approach, behind the scenes a *cell relation*



```
3
```

```
if N < 2 then 1
else N *  56_Factorial:Answer
```

```
FACTORIAL:N - 1
```

```
if N < 2 then 1
else N *  56_Factorial:Answer
```

56_FACTORIAL:ANSWER = Answer on the copy of Factorial whose N = (56_FACTORIAL:N + (- 1))

Figure 2. Forms/3 Factorial recursive spreadsheet. The user first created a spreadsheet Factorial (at left), then copied it and changed cell N's formula on the copy to Factorial:N–1 (at right). Finally the user entered the formula on Factorial:Answer (left) and the system automatically created any other copies needed to calculate the results.

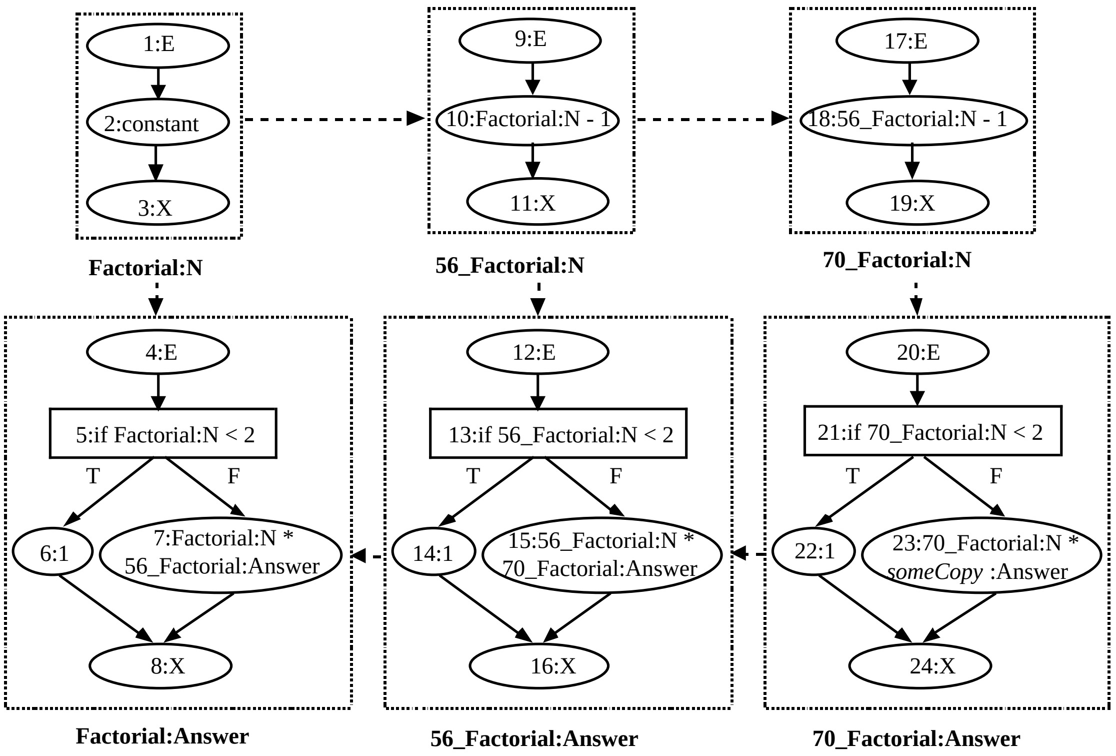


Figure 3. Cell relation graph for factorial recursive spreadsheets under the Extended WYSIWYT approach. Dashed arrows indicate dataflow relationships between cells. Within the formula graphs, *E* indicates entry into a formula and *X* indicates exit. Note, on the 70_Factorial:Answer, *someCopy*:Answer means the system doesn't have such a cell.

graph (CRG) is used. The CRG is used as both a theoretical model and an implementation device. A CRG is a pair (*V*, *E*) modeling the spreadsheet, 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* (see Figure 3). A formula graph models flow of control within a single cell's formula, and is comparable to a control flow graph. In the simplest non-recursive spreadsheets, there is one formula graph per cell; however, in spreadsheets in which some formulas have been replicated/shared, some sharing of formula graphs takes place [4].

This CRG model has been used to define several test adequacy criteria [13, 14]. The strongest criterion we have defined, *du-adequacy*, is the criterion we use in this paper to define when a spreadsheet has been tested "enough". Under this criterion, a cell *X* will be said to have been tested enough when all of its *definition-use associations* (abbreviated *du-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 constants 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. For example, nodes 7 and 15 in the figure are definitions. Node 7 also refers to (uses) node 15, and thus (15,7) is one du-association that needs to be tested. We assume that cells whose formulas are simply constant values, such as Factorial:N in Figure 2 and 3, do not need testing, and refer to them as "input cells" in this paper.

In the Extended WYSIWYT approach, information on recursive copies is kept individually for each copied cell, just as is true of the original, 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 WYSIWYT works on non-recursive spreadsheets.

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

Task 1: When a cell *C*'s formula is edited, static du-associations are collected for the cell, the collection of which is denoted *C.DUA*s.

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 a cell *C* is validated, *C.Trace* is consulted to find which of the du-associations in *C.DUAs* should be marked "covered."

Task 4: When a formula for a (non-input) producer of *C* is edited, *C.DUA*'s du-associations need to be reset to "not covered." (*C*'s *direct producers* are the cells *C* references; in general, we recursively define *C*'s *producers* as its direct producers and their producers.)

3.1.1 A Problem with Task 1

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.

We term a spreadsheet created from scratch by a user a *model spreadsheet*, and the cells on it *model cells*. These have a white background in Forms/3 such as the leftmost spreadsheet in Figure 2. Copies have gray backgrounds, except for cells whose formulas have been edited; since they now have their own formulas, these too are shown in white. In recursive spreadsheets, Forms/3 automatically generalizes what references to a spreadsheet copy mean [17]. If it did not, formulas such as the one for 56_Factorial:Answer in the copied spreadsheet in Figure 2 would be circular. 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.

In Figure 2, 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. 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) on another copy 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.

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 (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
```

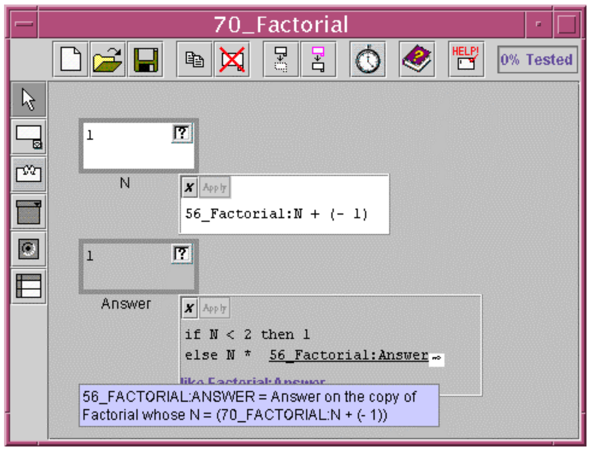
Figure 4. Collecting incoming du-associations.

Algorithm *CollectIncomingAssoc* of Figure 4 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 *RebuildAssoc* (Figure 4) 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. The time cost does not change from that of the original WYSIWYT version of this algorithm, which is analyzed in [13].

3.1.2 Infeasible DU-Associations

Even without recursion 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 termed *infeasible*. It is well known that infeasible elements such as these present a problem for testing methodologies, and we do not propose a comprehensive solution, but we need to avoid exacerbating the problem in dealing with recursion.

Unfortunately, the Extended WYSIWYT approach does exacerbate the problem, because the temporary du-associations solution (Section 3.1.1) is not sufficient for cells on spreadsheet copies that compute base



```
1
56_Factorial:N + (- 1)
1
if N < 2 then 1
else N *  56_Factorial:Answer
```

56_FACTORIAL:ANSWER = Answer on the copy of Factorial whose N = (70_FACTORIAL:N + (- 1))

Figure 5. 70_Factorial is the copy of the spreadsheet in Figure 2 that computes the base condition.

conditions. For example, 70_Factorial:Answer's reference in Figure 5 to another copy of Factorial will never be created or executed because 70_Factorial:Answer computes the base (see node 23 in Figure 3).

Even if the user decides to change N to a larger input value, the problem does not go away. Although it is solved for 70_Factorial, it re-materializes for whatever new copies the system automatically creates to compute the base condition. Thus, there will always be infeasible static du-associations under the Extended WYSIWYT approach, and 100% testedness of all viewable spreadsheets is unattainable.

3.2 Copy Representative Approach

Another possible approach was inspired by the Region Representative approach we developed to remove the tedium of testing groups of cells with replicated formulas [4]. The idea behind the Region Representative approach was to share most of the testing data among these similar cells. Recursive spreadsheets also introduce similar cells via copied formulas. 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 logical for copies of spreadsheets to share testing data.

3.2.1 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 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 the Figure 6. Further, as with the Region Representative approach, all copies of input cells (with constant formulas) are

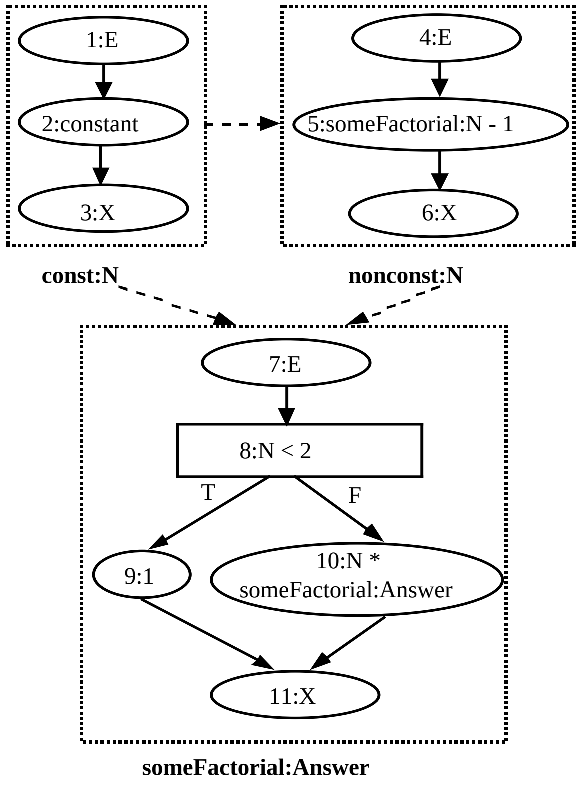


Figure 6. Cell relation graph for factorial recursive spreadsheets under the Copy Representative approach.

represented by a single formula graph.

The CRG in this figure is much smaller than the CRG in Figure 3. 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. For example, if the user validated Factorial:Answer, all Answer cells in Factorial's copies would also be validated.

3.2.2 Algorithms

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, and can point to an existing formula graph.

The Copy Representative algorithms' time costs presented below are the same per formula graph as the Extended WYSIWYT's, but the Copy Representative approach produces fewer formula graphs since some of them are shared.

Task 1 (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 6, the top left

```
Algorithm CollectIncomingAssoc (SharedGraph)
For each use ∈ SharedGraph.Uses
  Let useCells = {copies sharing SharedGraph}
  Let allDefCells = { definition cells that
       contribute to use }
  Let defSharedGraphs = {defCell.SharedGraph
      | defCell ∈ allDefCells}
  For each defSharedGraph ∈ defSharedGraphs
    For each def ∈ defSharedGraph.Defs
      Let DUA = (def, use, false)
      Add DUA to useSharedGraph.DUAs.Incoming
      Add DUA to defSharedGraph.DUAs.Outgoing
```

Figure 7. Collecting incoming du-associations under the Copy Representative approach.

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 *CollectIncomingAssoc* in Figure 7 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, that cell's *SharedGraph* is passed to algorithm *CollectIncomingAssoc*. 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 cells' formula graphs, and the use cells' formula graphs. With this algorithm, the system does not need the temporary du-associations of the Extended WYSIWYT approach.

Task 2 (collecting 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 formulas may execute different parts of that formula. For example, in Figure 2, cell Factorial:Answer executes the else-expression of the formula whereas Figure 5's 70_Factorial:Answer executes the then-expression. Execution traces are collected via an O(1) probe in the evaluation engine.

Task 3 (marking du-associations "covered" when the user validates cell C). Figure 8 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 *dynamicResolve* finds the concrete cell represented in this du-association in the context of *C* and recursively validates further.

Task 4 (resetting affected cells to "not covered" when the user edits a formula) is analogous to Task 1, and does not warrant separate discussion.

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

Figure 8. Algorithm for updating testedness following a validation.

4. Experiment

To guide our choice of method, we conducted an experiment to compare the two approaches. The specific objectives of the study were to investigate the following research questions:

RQ1. In which approach are spreadsheet programmers more effective in terms of du-adequacy?

RQ2. In which approach do spreadsheet programmers have fewer redundant test cases?

RQ3. In which approach are spreadsheet programmers more effective at finding faults?

RQ4. Which approach do spreadsheet programmers expect while testing?

4.1 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' 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 [7] and pilot studies.

4.2 Results

Brief summaries of the analyses of the data are provided below. For all analyses, $\alpha = .05$.

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. 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.

To address RQ2 (redundancy), for each subject we recorded the percentage of redundant test cases out of the total number of recorded test cases. A Repeated Measures ANOVA showed that the redundancy of the Copy Representative group was 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.

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 1).

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 correctly for the given approach.

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 (χ^2=.537, p >.1).

A Repeated Measures ANOVA also revealed that the Extended WYSIWYT group performed a greater number of tests than the Copy Representative group (F=8.52, df=1,45, p<.001), and also revealed a interaction between the testing approach and problems (F=3.68, df=2,44, p<.05).

	No faults	1 fault	2 faults
Copy Representative	2	7	15
Extended WYSIWYT	6	7	10

Table 1. The number of faults found in each group.

4.3 Discussion

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.

5. Related Work

There is little previous work outside of our own regarding testing of spreadsheets or in visual languages in general. The spreadsheet-oriented work has mostly focused on management devices to get users to test spreadsheets better, although there is also some work on comprehension aids for spreadsheet systems that might be useful for testing. Panko recently presented a summary of this work, and continues to update it [11].

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.

Some test adequacy criteria, however, do consider interactions among procedures. Dataflow test adequacy criteria are among these: Some interprocedural data dependence analysis techniques [8, 10] specifically calculate interprocedural du-associations. For such techniques, given a du-association, the all-uses dataflow test adequacy criterion [12] (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. In contrast to this, the two approaches presented in this paper both take context into account.

6. Conclusion

In this paper, we have presented two visual 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 has several testing-theoretic issues. However, its consistency with basic WYSIWYT could have caused it to be the 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, 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.

References

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

[2] M. Burnett and H. Gottfried, Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures, *ACM Trans. Computer-Human Interaction* 5(1), 1-33, Mar. 1998.

[3] 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, *J. Functional Programming* (to appear).

[4] M. Burnett, A. Sheretov, G. Rothermel, Scaling up a ‘what you see is what you test’ methodology to testing spread-sheet grids, *IEEE Symp. Vis. Lang.*, 30-37, Sept. 1999.

[5] E. Chi, J. Riedl, P. Barry, and J. Konstan, Principles for information visualization spreadsheets, *IEEE Computer Graphics and Applications*, July/Aug. 1998.

[6] 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.

[7] 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 Symp. Vis. Lang.*, Seattle, WA, 172-179, Sept. 2000.

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

[9] B. Myers, Graphical techniques in a spreadsheet for specifying user interfaces, *ACM Conf. Human Factors in Computing Systems*, 243-249, May 1991.

[10] H. D. Pande, B. G. Ryder and W. Landi, Interprocedural Def-Use Associations in C programs, *IEEE Trans. Software Eng.* 20(5), 385-403, May 1994.

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

[12] S. Rapps, and E. Weyuker, Selecting software test data using data flow information, *IEEE Trans. Software Eng.* 11, 367-375, Apr. 1985.

[13] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov, A methodology for testing spreadsheets, *ACM Trans. Software Eng. and Methodology*, (to appear).

[14] G. Rothermel, L. Li, C. DuPuis, and M. Burnett, What you see is what you test: A methodology for testing form-based visual programs, *Intl. Conf. Software Eng.*, 198-207, Apr. 1998.

[15] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, and G. Rothermel, An empirical evaluation of a methodology for testing spreadsheets, *Intl. Conf. Software Eng.*, 230-239, June 2000.

[16] T. Smedley, P. Cox, and S. Byrne, Expanding the utility of spreadsheets through the integration of visual programming and user interface objects, *ACM Wkshp. Advanced Visual Interfaces*, 148-155, May 1996.

[17] S. Yang and M. Burnett, From concrete forms to generalized abstractions through perspective-oriented analysis of logical relationships, *IEEE Symp. Vis. Lang.*, St. Louis, MO, 6-14, Oct. 1994.