Integrating Automated Test Generation into the WYSIWYT Spreadsheet Testing Methodology

MARC FISHER II and GREGG ROTHERMEL
University of Nebraska Lincoln
and
DARREN BROWN, MINGMING CAO, CURTIS COOK,
and MARGARET BURNETT
Oregon State University

Spreadsheet languages, which include commercial spreadsheets and various research systems, have had a substantial impact on end-user computing. Research shows, however, that spreadsheets often contain faults. Thus, in previous work we presented a methodology that helps spreadsheet users test their spreadsheet formulas. Our empirical studies have shown that end users can use this methodology to test spreadsheets more adequately and efficiently; however, the process of generating test cases can still present a significant impediment. To address this problem, we have been investigating how to incorporate automated test case generation into our testing methodology in ways that support incremental testing and provide immediate visual feedback. We have used two techniques for generating test cases, one involving random selection and one involving a goal-oriented approach. We describe these techniques and their integration into our testing environment, and report results of an experiment examining their effectiveness and efficiency.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools* (*e.g., data generators, coverage testing*); D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*; H.4.1 [**Information Systems Applications**]: Office Automation—*Spreadsheets*

General Terms: Algorithms, Experimentation, Verification

This article is a revised and expanded version of a work presented at the International Conference on Software Engineering, May, 2002 [Fisher et al. 2002].
This work has been supported by the National Science Foundation by ESS Award CCR-9806821 and ITR Award CCR-0082265 to Oregon State University.
Authors' addresses: M. Fisher II and G. Rothermel, Department of Computer Science and Engineering, University of Nebraska Lincoln, 256 Avery Hall, Lincoln, NE 68588; email: {mfisher, grother}@cse.unl.edu; D. Brown, M. Cao, C. Cook, and M. Burnett, School of Electrical Engineering and Computer Science, Oregon State University, 1148 Kelley Engineering Center, Corvallis, OR 97331; email: browndar@lifetime.oregonstate.edu, cmm@us.ibm.com,{cook,burnett}@eecs.oregonstate.edu.

Additional Key Words and Phrases: End-user software engineering, testing, test case generation, end-user programming

1. INTRODUCTION

Spreadsheets are used by a wide range of end users to perform a variety of important tasks, such as managing retirement funds, performing tax calculations, forecasting revenues, managing payrolls, and even assessing the quality of batches of pharmaceutical products. The spreadsheet language paradigm is also a subject of ongoing research; for example, there is research into using spreadsheet languages for matrix manipulation problems [Viehstaedt and Ambler 1992], for scientific visualization [Chi et al. 1997], for providing steerable simulation environments for scientists [Burnett et al. 1994], and for specifying full-featured GUIs [Myers 1991; Smedley et al. 1996].

It is important that spreadsheets function correctly, but experimental and anecdotal evidence [Panko 1995, 1998] shows that spreadsheets often contain faults, and that these faults can have severe consequences. For example, an erroneous spreadsheet formula inflated the University of Toledo's projected annual revenue by 2.4 million dollars, requiring sudden budget cuts [Smith 2004]. Another spreadsheet formula error caused the stocks of Shurgard Inc. to be devalued after two employees were overpaid by $700,000 [Scott 2003], and a cut-and-paste error in a bidding spreadsheet cost the Transalta Corporation 24 million dollars through overbidding [Cullen 2003]. Compounding these problems is the unwarranted confidence spreadsheet users tend to have in the correctness of their spreadsheets [Brown and Gould 1987; Wilcox et al. 1997].

In spite of such evidence, little work had been done until recently to help end users assess the correctness of their spreadsheets. Thus, we have been working on software engineering methods that can be used by end users in spreadsheet programming environments [Burnett et al. 2004]. One of the main components of this effort is termed the "What You See Is What You Test" (WYSIWYT) methodology [Rothermel et al. 2001, 1997, 1998]. The WYSIWYT methodology provides feedback about the "testedness" of cells in spreadsheets in a way that is incremental, responsive, and entirely visual. Empirical studies have shown that this methodology can help end users test their spreadsheets more adequately and more efficiently, and detect faults in their spreadsheets [Krishna et al. 2001; Rothermel et al. 2000]. In subsequent work, we have extended the approach to apply to large grids of cells with shared formulas [Burnett et al. 1999, 2002] and to incorporate the re-use of test cases [Fisher et al. 2002].

As presented in the aforementioned articles, the WYSIWYT methodology has relied solely on the judgments of spreadsheet users in the creation of test cases for spreadsheets. In general, the process of manually creating appropriate test cases is laborious and its success depends on the experience of the tester. This problem is especially serious for users of spreadsheet languages, who typically are not experienced programmers and lack background in testing. Existing research on automated test generation (e.g., [Clarke 1976; DeMillo and Offutt 1991; Ferguson and Korel 1996; Gotlieb et al. 1998; Korel 1990a; Korel 1990b;

Ramamoorthy et al. 1976]), however, has been directed at imperative languages, and we can find no research specifically addressing automated test generation for spreadsheet languages.

To address this problem, we have been investigating how to automate the generation of test inputs for spreadsheets in ways that support incremental testing and provide immediate visual feedback. We have incorporated two techniques for test generation into the WYSIWYT methodology: One using randomly chosen inputs and one using a dynamic, goal-oriented approach [Ferguson and Korel 1996]. In this article, we describe these techniques and their integration into WYSIWYT, and we report results of experiments examining their effectiveness and efficiency. The results show that the goal-oriented technique is highly effective at generating useful test inputs, and typically much more effective than the random technique. Providing range information to inform random generation, however, improves the prospects for that technique, while doing little for the goal-oriented technique. Both techniques can be applied at various levels of focus with respect to spreadsheet elements, ranging from individual cell interactions, to individual cells, to the entire spreadsheet.

The remainder of this article is organized as follows. Section 2 presents background material on the spreadsheet programming paradigm, the WYSIWYT testing methodology, and test generation techniques for imperative programs. Section 3 discusses the requirements for an automatic test generation technique for spreadsheet languages, and then presents our techniques. Section 4 presents the design and results of the experiments we have performed to investigate our testing methodology, and discusses the implications of those results. Finally, Section 5 presents conclusions and discusses future work.

2. BACKGROUND

2.1 Spreadsheet Languages and Forms/3

Users of spreadsheet languages "program" by specifying cell formulas. Each cell's value is defined by that cell's formula, and as soon as the user enters a formula, it is evaluated and the result is displayed. The best-known examples of spreadsheet languages are found in commercial spreadsheet systems, but there are also many research systems (e.g. [Burnett et al. 2001; Chi et al. 1997; Leopold and Ambler 1997; Smedley et al. 1996]) based on this paradigm.

In this work we utilize Forms/3 [Burnett et al. 2001], a Turing-complete research spreadsheet language. We do this because we have access to the source code for the Forms/3 programming environment and can therefore easily implement our methodologies in that environment, together with instrumentation that lets us monitor the results of experiments using these methodologies. Further, this approach lets us consider features found in commercial spreadsheet languages, but will also allow us to eventually consider new features of spreadsheet languages that are currently under investigation but not yet supported in commercial languages.

To be precise, the spreadsheet language model that we consider is a "pure" spreadsheet model (and a subset of Forms/3) that includes ordinary

```
formula ::= BLANK | expr
expr ::= CONSTANT | CELLREF | infixExpr   |
         prefixExpr | ifExpr | composeExpr
infixExpr ::= subExpr infixOperator subExpr
prefixExpr ::= unaryPrefixOperator subExpr   |
               binaryPrefixOperator subExpr subExpr
ifExpr ::= IF subExpr THEN subExpr ELSE subExpr
composeExpr ::= COMPOSE subExpr withclause
subExpr ::= CONSTANT | CELLREF | (expr)
infixOperator ::= +| − | ∗ |/| MOD | AND | OR | = | < | <= | > | >=
unaryPrefixOperator ::= NOT | CIRCLE | ROUND
binaryPrefixOperator ::= LINE | BOX
withclause ::= WITH subExpr AT (subExpr subExpr) |
               WITH subExpr AT (subExpr subExpr) withclause
```

Fig. 1. Grammar for formulas considered in this research. A cell with no formula is equivalent to a cell with formula *BLANK*; the result of evaluating such a formula is a distinguished value, NOVALUE.

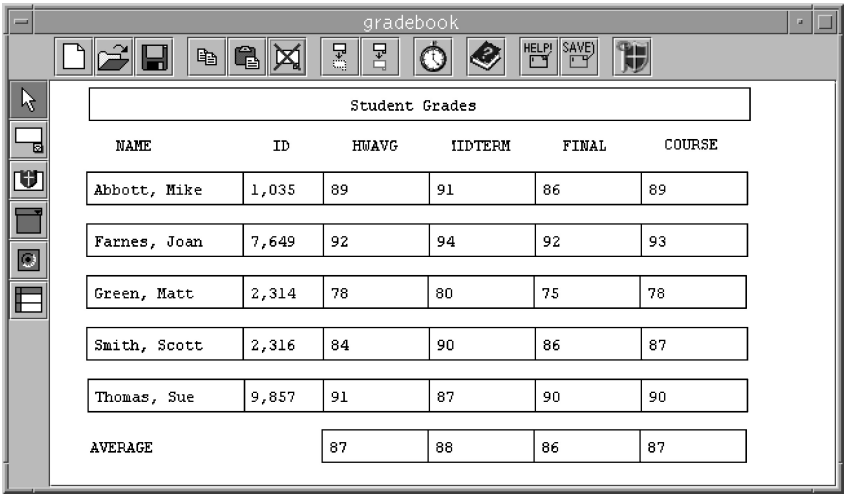

Fig. 2. Forms/3 spreadsheet for calculating grades.

spreadsheet-like formulas such as those described by the grammar given in Figure 1, but excludes advanced programmer-oriented features such as macros, imperative sublanguages, indirect addressing, and recursion. The subset uses ordinary spreadsheet formulas for both numeric and graphical computations; the figures presented in this article and the spreadsheets utilized in the experiments it describes were programmed using this subset.

Figure 2 shows a traditional-style spreadsheet used to calculate grades in Forms/3. The spreadsheet lists several students and several assignments performed by them. 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 cell gives the overall course average (formulas not shown).

Figure 3 shows a second example of a Forms/3 spreadsheet, `Budget`, that calculates how many pens and paper clips an office will have after an order and whether that order is within a given budget amount.

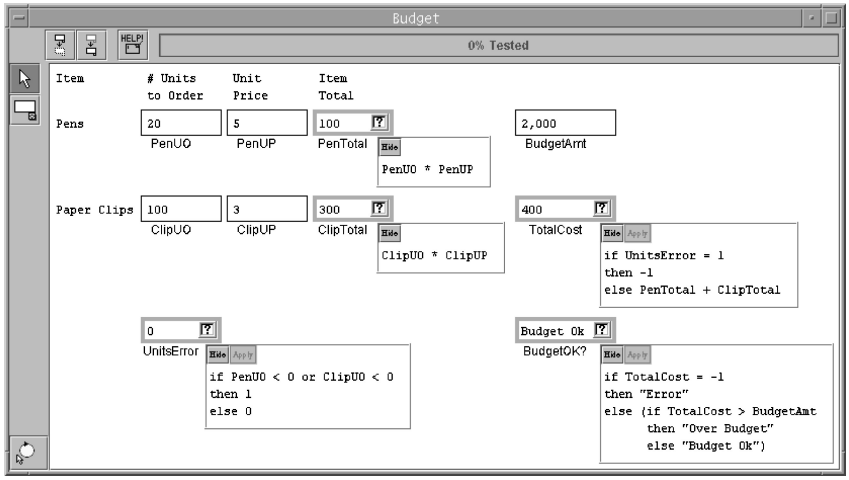

Fig. 3. Forms/3 spreadsheet Budget.

As the figures show, Forms/3 spreadsheets, like traditional spreadsheets, consist of cells, but these cells can be placed arbitrarily on the screen. Also, in Figure 3 cell formulas are displayed, but in general, the user can display or hide formulas. As the spreadsheets (and the grammar in Figure 1) illustrate, the only dependencies between one cell and another are data dependencies. Because of this fact, cells can be evaluated in any order that preserves these dependencies.

2.2 The WYSIWYT Methodology

In our "What You See Is What You Test" (WYSIWYT) methodology for testing spreadsheets [Rothermel et al. 1998, 2000, 2001], as a user incrementally develops a spreadsheet, he or she can also test that spreadsheet incrementally. As the user changes cell formulas and values, the underlying engine automatically evaluates cells, and the user indicates whether the results displayed in those cells are correct. Behind the scenes, these indications are used to measure the quality of testing in terms of a dataflow adequacy criterion that tracks coverage of interactions between cells caused by cell references (other applicable criteria, and our reasons for focusing on this one, are presented in Rothermel et al. [1997, 2001]; the primary reason involves the fact that a large percentage of spreadsheet faults involve cell reference errors made in formulas [Panko 1998], and dataflow interactions track cell references.)

The following example illustrates the process from the user's perspective. Suppose the user constructs the `Budget` spreadsheet by entering cells and formulas, reaching the state shown in Figure 3. At this point, all cells other than input cells have red borders (light gray in this article), indicating that their formulas have not been (in user terms) "tested" (input cells are cells whose formulas contain no references and are, by definition, fully tested; their borders are thin and black to indicate to the user that they are not testable.)

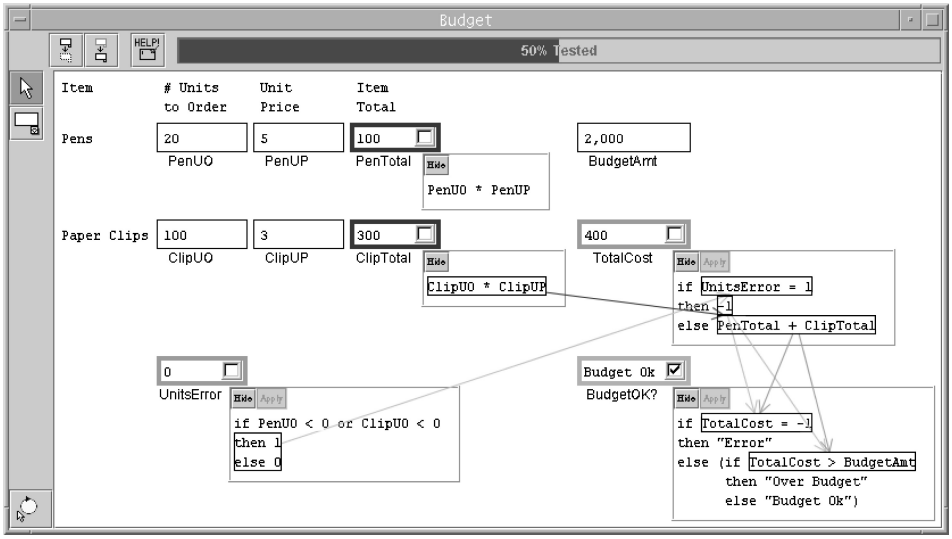

Fig. 4. Forms/3 spreadsheet Budget with testing information displayed after a user validation.

Suppose the user looks at the values displayed on the screen and decides that cell `BudgetOK?` contains the correct value, given the values currently present in input cells. To communicate this decision, the user checks off the value by clicking on the decision box in the upper-right corner of that cell. One result of this action, shown in Figure 4, is the appearance of a checkmark in the decision box, communicating that the cell's output has been indicated to be correct under the current assignment of values to input cells (two other decision box states, empty and question mark, are possible: each communicates that the cell's output has not been indicated to be correct under the current assignment of values to input cells. In addition, the question mark shows that clicking on a cell's decision box would increase "testedness.")

A second result of the user's action is that the colors of the `BudgetOK?` cell's borders become more blue, communicating that interactions caused by references in that cell's formula have been validated; that is, used in a computation that has resulted in the production of a cell value indicated to be correct by the user. In the example, in the formula for `BudgetOK?`, interactions caused by references in the `else` clause have now been validated, but interactions caused by references in the `then` clause have not. Thus, that cell's border is partially blue (slightly darker gray in this article). Testing results also flow "upstream" in the dataflow to other cells whose formulas have been used in producing a value the user has indicated to be correct. In our example, all interactions ending at references in the formula for cell `ClipTotal` have been validated; hence, that cell's border is now fully blue (very dark gray in this article).

If users choose, they can also view the interactions caused by cell references by displaying dataflow arrows between cells or between subexpressions in formulas; in the example, the user has chosen to view interactions originating and ending at cell `TotalCost`. These arrows depict testedness information at a finer

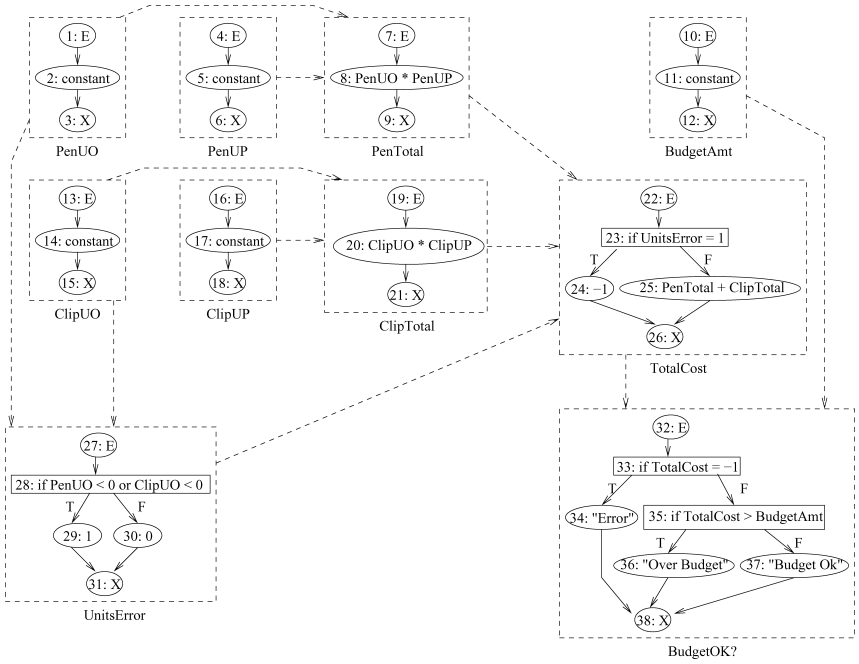

Fig. 5. Cell relation graph for Budget.

granularity than cell borders, following the same color scheme as the borders (the color differences do not show up well in this black-and-white figure).

If the user next modifies a formula, interactions potentially affected by this modification are identified by the system, and information on those interactions is updated to indicate that they require retesting. The updated information is immediately reflected in changes in the various visual indicators just discussed (e.g., replacement of blue border colors by less blue colors).

Although a user of our methodology need not be aware of it, the methodology is based on the use of a dataflow test adequacy criterion adapted from the *output-influencing-all-du-pairs* dataflow adequacy criterion defined for imperative programs [Duesterwald et al. 1992] (the work in Duesterwald et al. [1992] was, in turn, based on earlier work on dataflow adequacy criteria [Frankl and Weyuker 1988; Laski and Korel 1993; Rapps and Weyuker 1985]). For brevity, we call our adaptation of this criterion the *du-adequacy* criterion. We precisely define this criterion in Rothermel et al. [2001]; here, we summarize that presentation.

The du-adequacy criterion is defined via an abstract model of spreadsheets called a *cell relation graph* (CRG). Figure 5 shows the CRG for the spreadsheet `Budget`. A CRG consists of a set of *cell formula graphs* (enclosed in rectangles in the figure) that summarize the control flow within formulas, connected by edges (dashed lines in the figure) summarizing data dependencies between cells. Each cell formula graph is a directed graph, similar to a control flow graph for imperative languages, in which each node represents an expression in a cell formula and each edge represents flow of control between expressions.

There are three types of nodes: Entry and exit nodes, representing initiation and termination of the evaluation of the formula; definition nodes, representing simple expressions that define a cell's value; and predicate nodes, representing predicate expressions in formulas. Two edges extend from each predicate node: these represent the true and false branches of the predicate expression.

A *definition* of cell C is a node in C's formula graph representing an expression that defines C, and a *use* of C is either a *computational use* (a nonpredicate node containing an expression that refers to C) or a *predicate use* (an out-edge from a predicate node containing an expression that refers to C). A *definition-use association* (*du-association*) links a definition of C with a use of C. A du-association is considered *exercised* when inputs have been found that cause the expressions associated with its definition and its use to be executed. A du-association is considered *validated* when it is exercised, and either the value in the cell containing the use, or the value in a cell that dynamically uses (directly or transitively) the cell containing the use, has been indicated by a user to be correct. Under the du-adequacy criterion, testing is adequate when each du-association in a spreadsheet has been validated at least once.

In this model, a *test case* for a spreadsheet is a tuple (I,C), where I is a vector of values given to input cells in the spreadsheet, and C is a cell whose value the user has indicated to be correct under that input configuration. A *test* is an explicit decision by the user that C's value is correct given the current configuration I of input cell values, signaled by their clicking on C's decision box; in such a case, we refer to the user's action as a *correctness indication*, and we say that C has been *indicated to be correct*.

It is not always possible to exercise (and hence, not always possible to validate) all of the du-associations in a spreadsheet; those that cannot be exercised by any inputs are referred to as *infeasible du-associations*. In general, the problem of identifying such du-associations is undecidable [Frankl and Weyuker 1988; Weyuker 1993].

2.3 Automated Test Generation Techniques for Imperative Languages

There has been considerable research on techniques for automatic test generation for imperative languages. Depending on what aspect of these techniques is to be addressed, there are different ways to classify these techniques. An ATCG technique could be categorized according to the test adequacy criterion it uses, for example, statement coverage or branch coverage. Alternatively, ATCG techniques could be classified according to the software artifact on which they operate. For example, there has been work on generating tests from specifications [Chang and Richardson 1999; Marinov and Khurshid 2001; Offutt and Abdurazik 1999], from formal models [Avritzer and Weyuker 1995; Hartman and Nagin 2003; Visser et al. 2004], and from code [Baresel et al. 2004; Boyapati et al. 2002; Gotlieb et al. 1998; Gupta et al. 1998].

Another important classification involves the mechanism by which automated test generation techniques generate tests. Along these lines, Ferguson and Korel [1996] classify ATCG techniques into three categories: *random*, *path-oriented* and *goal-oriented*. We summarize that classification here.

Random test generation techniques [Bird and Munoz 1983] generate tests by randomly selecting input values.

Path-oriented test generation techniques first select a program path that will meet a testing requirement, and then attempt to find input values that cause that path to be executed. Ferguson and Korel distinguish two types of path-oriented techniques: those based on *symbolic execution*, and those that are *execution-oriented*. Techniques based on symbolic execution [Clarke 1976; DeMillo and Offutt 1991; Gotlieb et al. 1998; Offutt 1991] use symbolic execution to find the constraints, in terms of input variables, that must be satisfied in order to execute a target path, and attempt to solve this system of constraints. The solution provides a test case for that path. A disadvantage of these techniques is that they can waste effort attempting to find inputs for infeasible paths. They can also require large amounts of memory to store the expressions encountered during symbolic execution, and powerful constraint solvers to solve complex equalities and inequalities. They may also have difficulties handling complex expressions. Execution-oriented techniques [Korel 1990b] alleviate some of these difficulties by incorporating dynamic execution information into the search for inputs, using function minimization to solve subgoals that contribute toward an intended coverage goal.

Goal-oriented test generation techniques [Ferguson and Korel 1996; Korel 1990a], like execution-oriented techniques, are also dynamic, and use function minimization to solve subgoals leading toward an intended coverage goal. However, goal-oriented techniques focus on the final goal rather than on a specific path, concentrating on executions that can be determined (e.g., through the use of data dependence information) to possibly influence progress toward that goal. Like execution-oriented techniques, goal-oriented techniques take advantage of the actual variable values obtained during execution to try to solve problems with complex expressions; however, by not focusing on specific paths, the techniques gain effectiveness [Ferguson and Korel 1996] (these distinctions, and the characteristics of goal-oriented techniques, will be further clarified in Section 3.2 when we present the chaining technique, a goal-oriented technique that is used in this research).

More recently, an additional class of test generation techniques not mentioned by Ferguson and Korel has emerged, in the context of *evolutionary testing* [Baresel et al. 2004; Michael et al. 2001]. These techniques use genetic algorithms to generate tests, focusing on goals rather than specific paths.

3. AUTOMATED TEST GENERATION AND WYSIWYT

In any testing methodology that does not automatically generate tests, the users themselves must generate useful test inputs, and this can be difficult, particularly if those users are end users. To help with this, we have developed a test generation methodology for spreadsheets, integrated support for that methodology into our WYSIWYT approach, and implemented that methodology in Forms/3. To present our methodology, we begin by describing the user actions and system responses that comprise the basic version of that methodology using random input generation. Then, in Section 3.2, we show how to extend this

```
framework GenerateTest(Cells, Arrows)
inputs    Cells : Cells indicated by user
          Arrows : Arrows indicated by user
1.  UsefulDUs = CalculateUsefulDUs(Cells, Arrows)
2.  InputCells = CalculateInputCells(UsefulDUs)
3.  InputValues = SaveInputCellValues(InputCells)
4.  if GenerateTest(UsefulDUs, InputCells) then
5.    UpdateDisplay()
6.  else
7.    RestoreConfig(InputCells,InputValues)
8.  end if
```

Fig. 6. An algorithm framework for supporting dynamic ATCG techniques in the WYSIWYT context.

approach to incorporate Ferguson and Korel's goal-oriented input generation technique, in Section 3.3 we discuss a refinement involving provision of range information with spreadsheet cells, and in Section 3.4 we discuss requirements for extending our test generation methodologies to noninteger datatypes.

3.1 Basic Methodology

Suppose a user desires help in increasing the testedness of a spreadsheet. With our methodology, a user may select any combination of cells or du-association arrows on the visible display (or, selecting none, signal interest in all cells and arrows in the spreadsheet), then push the "Help Me Test" button in the Forms/3 toolbar. At this point, the underlying test generation system responds, attempting to generate a test input.

Figure 6 presents an algorithm framework for supporting dynamic ATCG techniques in the WYSIWYT context. In the approach presented, the system's first task (line 1) is to call `CalculateUsefulDUs` to determine the set *UsefulDUs* of du-associations relevant to the user's request: that is, the du-associations in the area of interest that have not been validated. `CalculateUsefulDUs` proceeds as follows. If the user has not selected specific cells or du-association arrows, *UsefulDUs* is the set of all unvalidated du-associations in the spreadsheet. If the user has selected one or more cells in the spreadsheet, *UsefulDUs* includes each unvalidated du-association that has its use in one of those cells. Finally, if the user has selected one or more du-association arrows in the spreadsheet, *UsefulDUs* includes each of the unvalidated du-associations associated with each such arrow (in our interface, each du-association ending at a computational use is represented by a single arrow ending at the subexpression associated with that computational use, whereas the pair of du-associations ending at the true and false predicate-use edges flowing from some predicate node are represented by a single arrow ending at the subexpression associated with that predicate use).

The second task of the system (line 2) is to call `CalculateInputCells` to determine the set of input cells, *InputCells*, that can potentially cause du-associations in *UsefulDUs* to be exercised; these are the cells whose values a test generation technique can profitably manipulate. Because this information is maintained by spreadsheet evaluation engines to perform updates following cell edits (this is true of most other spreadsheet languages as well as of Forms/3

[Rothermel et al. 2001]), it is available in data structures kept by the engine, and `CalculateInputCells` simply retrieves it from there.

Given *UsefulDUs* and *InputCells*, the test generation system can attempt to generate a test input. The system first saves the existing configuration of input cell values (line 3) for restoration if generation fails, and then, through `GenerateTest` invokes a test generation technique (line 4). As Section 2.3 indicated, there are many test generation techniques that could be utilized; the simplest of these is to randomly generate input values. To provide this technique, we have `GenerateTest` invoke a random generator, `Random`.

`Random` randomly assigns values to the cells in *InputCells*, invokes the spreadsheet's evaluation engine to cause the effects of those values to be propagated throughout the spreadsheet, and determines whether the evaluation causes any du-association in *UsefulDUs* to be exercised. If no such du-association is exercised, `Random` repeats this process with a new set of random values, iterating until either a set of values that exercises a du-association of interest is found, or a built-in time limit is reached. As the system applies these new input values, the values appear in the spreadsheet itself. Displaying these values carries a performance penalty, but an advantage is that it communicates to the user what the system is doing, an understanding of which is often a significant factor in users' effectiveness and continuing use of a system [Belkin 2000; Corritore et al. 2001].

If `Random` exercises a du-association in *UsefulDUs*, the system has generated a potentially useful set of test inputs; however, this is not yet a test—recall that a test also includes a user's correctness indication. Hence, the system now needs to communicate to the user which cell(s) that are affected by this set of inputs could now usefully be indicated to be correct. To direct the user's attention to such cells, the test generation system determines the set of relevant validatable output cells resulting from the new test inputs, and makes these visible to the user by displaying them with question marks in their decision boxes. A *relevant validatable output cell* is any cell such that, if its value is indicated to be correct by a user, will cause the system to mark some du-association in *UsefulDUs*—the set of du-associations that are coverage targets of interest to the user—to have been validated. For example, if the user requested help testing cell `BudgetOK?` in Figure 4, the system would manipulate the values in cells `PenUO`, `PenUP`, etc., and would present `BudgetOK?` as a relevant validatable output cell.

If the system succeeds in finding a potentially useful set of test inputs and displaying relevant validatable output cells, the user at this point can indicate the value of one such cell to be correct. Alternatively, the user can ignore the generated values (e.g., if he or she dislikes the set of inputs generated) and choose to try again with the same or different cells or du-association arrows selected, in which case the test generation technique tries again using new seeded values.

If, on the other hand, the system reaches its built-in time limit without finding a useful set of test inputs, it restores the previous input state (line 7) and tells the user that it has been unable to generate useful test inputs. In this case, too, the user can try again and the system will begin with new seeded values.

```
algorithm Chaining(UsefulDUs,InputCells)
input     UsefulDUs : a list of du-associations to try to cover
          InputCells : a list of input cells relevant to UsefulDUs
returns   success or failure
1.  for each (d,u) ∈ UsefulDUs do
2.    donewiththisDU = false
3.    while not donewiththisDU do
4.      b = FindBreakPoint(d,u)
5.      MinimizeBreakPoint(b)
6.      if not Satisfied(b) then
7.        Chaining(ChainDUs(b),InputCells)
8.      end if
9.      if not Satisfied(b) then
10.       donewiththisDU = true
11.     else if Exercised(d,u) then
12.       return success
13.     end if
14.   end while
15. end for
16. return failure
```

Fig. 7. Chaining algorithm.

3.2 Goal-Oriented Test Generation

Random is easy to implement and gave us a way to quickly prototype our methodology. Moreover, it was suggested that Random might be sufficient for spreadsheets, since most spreadsheets do not use loops, aliasing, or other language features that complicate test generation for imperative programs.[1] On the other hand, spreadsheets and the WYSIWYT approach lend themselves naturally to goal-oriented test generation that requires dynamic execution information (already tracked by WYSIWYT) and the ability to quickly re-execute a program under various inputs (already provided by a spreadsheet evaluation engine). We therefore also investigated goal-oriented test generation techniques. Ultimately, we adapted Ferguson and Korel's "Chaining Approach" [1996] to our purpose; in this article, to we refer to our adaptation as Chaining.

3.2.1 *Overall Algorithm.* Figure 7 presents the Chaining Algorithm. Like Random, Chaining is invoked to find a set of inputs that causes one or more du-associations in *UsefulDUs* to be exercised. In terms of Figure 6, we simply have GenerateTest (line 4) invoke Chaining; however, whereas Random simply generates inputs for all cells in *InputCells* and then checks whether any du-association in *UsefulDUs* is exercised, Chaining iterates through *UsefulDUs* considering each du-association in turn. On finding a useful input set, Chaining terminates, and the visual devices described above for indicating relevant validatable output cells are activated. If Chaining fails on all du-associations in *UsefulDUs*, then, like Random, it indicates this to the system, which reports that it could not find a test input.

We now describe the process by which, in considering a du-association (d,u), Chaining proceeds. In spreadsheets, the problem of finding input values to

[1]Personal communication, Jeff Offutt.

exercise (d,u) can be expressed as the problem of finding input values that cause both the definition d and the use u to be executed (an additional requirement present for imperative programs—that the definition "reaches" the use—is achieved automatically in spreadsheets of the type we consider, provided the definition and use are both executed, since these spreadsheets do not contain loops or "redefinitions" of cells [Rothermel et al. 2001]). For example, to exercise du-association (24,(33,T)) in `Budget` (see the CRG in Figure 5 and its associated spreadsheet in Figure 4), input values must cause node 24 in the formula graph for `TotalCost` to be reached, and they must also cause the true branch of node 33 in the formula graph for cell `BudgetOK?` to be reached.

The conditions that must be met within a single formula to execute d (or u) can be expressed in terms of constraint paths. Let F be the cell formula graph for a cell containing definition d (or use u), and let e be F's entry node. The *constraint path for* d (or u) is the sequence of nodes and branches in F, beginning with e, that fall along the path through F from e to d (or to u). For example, the constraint paths for definition 24 and use (33, T) in the CRG for `Budget` are (22,23,(23,T),24) and (32,33,(33,T)), respectively.

Given constraint paths p and q for definition d and use u, respectively, the *constraint path for du-association* (d,u) consists of the concatenation of p and q. Thus, for example, the constraint path for du-association (24,(33,T)) in `Budget` is (22,23,(23,T),24,32,33,(33,T)).

When considering du-association (d,u), `Chaining` first constructs the constraint path p for (d,u). Given our methodology, it is necessarily the case that under the current assignment of values to input cells, (d,u) is not exercised—otherwise, it would not be included in *UsefulDUs*. Thus, it must be the case that under the current inputs, one or more predicates in p are being evaluated in a manner that causes other nodes or branches in p (and ultimately, d and/or u) not to be reached. `Chaining`'s task is to alter this situation by finding an assignment of values to input cells that causes all nodes and branches in p to be reached.

To do this, `Chaining` compares the constraint path p for (d,u) to the sequence of nodes and branches listed in the current execution traces for the cells containing d and u. The *execution trace* for a cell C is the sequence of nodes and branches, in the formula graph for C, reached during the most recent evaluation of C; execution traces can be retrieved from the spreadsheet engine, which already collects them for use by the WYSIWYT subsystem. `Chaining` then identifies the break point in p, described informally as the predicate controlling the earliest "incorrectly taken branch" in the concatenated execution traces for the cells containing d and u. More formally, given constraint path p for (d,u), and given e, the sequence of nodes and branches listed in the *execution traces* for the cells containing d and u, the *break point* in p is the first predicate node n in p such that the successor of n in p differs from the successor of n in e.

In the example we have been considering, the concatenated execution traces, assuming the spreadsheet's input cells have the values shown in Figure 4, are (22,23,(23,F),25,26,32,33,(33,F),35,(35,F),37,38), and thus, the break point is predicate node 23. In Figure 7, the calculation of the constraint path for (d,u)

and of the break point in that constraint path occurs in the FindBreakPoint function invoked in line 4.

Given a break point b, Chaining's next task is to find inputs that cause b to take the opposite branch. To do this, the technique uses a constrained linear search procedure over the input space (MinimizeBreakPoint, line 5); we describe this procedure in Section 3.2.2. If this search procedure fails to cover the break point, b is designated a *problem break point*, and the algorithm attempts to "chain back" from b, a procedure described shortly. If both the search and the "chaining" process fail, the algorithm has failed to cover the target du-association, and it moves on to the next du-association in *UsefulDUs* (lines 9–10).

If the break point is satisfied, two outcomes are possible:

(1) Du-association (d,u) is now exercised. The technique has succeeded and terminates (lines 11–12);
(2) Inputs that cause the desired branch from the predicate b have been found, but a subsequent predicate on the constraint path has not taken the right direction (i.e., another break point exists), so (d,u) has not yet been executed. In this case, Chaining repeats the process of finding a break point and initiating a constrained linear search, attempting to make further progress.

(A variation on this algorithm lets Chaining report success on executing *any* du-association in *UsefulDUs*, an event that can occur if *UsefulDUs* contains more than one du-association and if, in attempting to execute one specific du-association, Chaining happens on a set of inputs that execute a different du-association in *UsefulDUs*. The results of this variation may make sense from an end user's point of view, because the fact that Chaining iterates through du-associations is incidental to the user's request: the user requested only that some du-association in a set of such du-associations be executed. Our prototype in fact implements this variation; however, to simplify the presentation we focus here on the single du-association iterated on.)

In the example we have been considering, the only outcomes possible are that the search fails, or that it succeeds, causing du-association (24, (33, T)) to be exercised. If, however, cell TotalCost had contained another predicate node p in between nodes 23 and 24 such that node 24 is reached only if p evaluates to "false," then it could happen that the inputs found to cause 23 to evaluate to "true" did not also cause p to evaluate to "false," in which case the while loop would continue, and b would now be p.

When a problem break point is encountered, Chaining cannot make progress on that break point. It is possible, however, that by exercising some other du-association that influences the outcome of the predicate in the break point, Chaining will be able to make progress. Thus, faced with a problem break point b, Chaining calls ChainDUs (line 7), which collects a set $ChainDUs$ of other du-associations (d',u') in the spreadsheet that have two properties: (1) u' is the predicate-use associated with the alternate branch from b, (i.e., the branch we wish to take), and (2) d' is not currently exercised. These du-associations, if exercised, necessarily enable the desired branch to be taken. Chaining iterates

```
algorithm MinimizeBreakPoint(b)
input     b : a break point to cover

 1.  finished = false
 2.  delta = 0
 3.  f = CalculateBranchFunction(b)
 4.  while not finished do
 5.    finished = true
 6.    for each i ∈ Inputs(b) do
/* Exploratory moves */
 7.      v = CurrentValue(f)
 8.      i = i + 1
 9.      if CurrentValue(f) > v then
10.        delta = 1
11.      else
12.        i = i − 2
13.        if CurrentValue(f) > v then
14.          delta = −1
15.        else
16.          i = i + 1
17.          delta = 0
18.        end if
19.      end if
/* Pattern Moves */
20.      while delta ≠ 0 do
21.        finished = false
22.        v = CurrentValue(f)
23.        if v > 0 then
24.          return
25.        end if
26.        i = i + delta
27.        if CurrentValue(f) > v then
28.          delta = delta × 2
29.        else
30.          i = i − delta
31.          if |delta| = 1 then
32.            delta = 0
33.          else
34.            delta = delta ÷ 2
35.          end if
36.        end if
37.      end while
38.    end for
39.  end while
```

Fig. 8. Algorithm for minimizing the branch function for a break point.

through du-associations in $ChainDUs$, applying (recursively) the same process described for use on (d,u) to each (as discussed in Ferguson and Korel [1996], a bound can be set on the depth of this recursion to limit its cost; however, we did not set such a bound in our prototype).

In our example, assume that node 23 is designated a problem break point (i.e., the search algorithm was not able to force execution at node 23 to take the "true" branch). In this case, `Chaining` would proceed to "chain back" from node 23. It would then set $ChainDUs$ to (29,(23,T)) and try to exercise this du-association.

3.2.2 *The Search Procedure.* The search procedure used by `Chaining` to find inputs that cause predicates to take alternative branches involves two steps. First, a *branch function*—a real-valued function expressing the conditions that must be met in order to cause a predicate to take a certain outcome—is created, based on the predicate, to guide the search. Second, a sequence of input values are applied to the spreadsheet in an attempt to satisfy the branch function. We describe these steps in the following text and in Figure 8.

A branch function should have two characteristics. First, changes in the values of the branch function as different inputs are applied should reflect changes in closeness to the goal. Second, the rules used to judge whether a branch function is improved or satisfied should be consistent across branch functions; this allows branch functions to be combined to create functions for complex predicates. To satisfy these criteria, we defined branch functions for relational operators in spreadsheets similar to those presented in Ferguson and Korel [1996], as shown in Table I. With these functions: (1) if the value of the branch function is less than or equal to 0, the desired branch is not exercised; (2)

Table I. Branch Functions for True Branches of Relational Operators

Relational Operator	Branch Function
$l < r$	$r - l$
$l > r$	$l - r$
$l \leq r$	if $r - l \geq 0$ then $r - l + 1$ else $r - l$
$l \geq r$	if $l - r \geq 0$ then $l - r + 1$ else $l - r$
$l = r$	if $l = r$ then 1 else $-\lvert l - r \rvert$
$l \neq r$	$\lvert l - r \rvert$

Table II. Branch Functions for Logical Operators

Logical Operator	Branch Function
l and r	True branch: if $f(l, true) \leq 0$ and $f(r, true) \leq 0$ then $f(l, true) + f(r, true)$ else $\min(f(l, true), f(r, true))$ False branch: if $f(l, false) \leq 0$ and $f(r, false) \leq 0$ then $f(l, false) + f(r, false)$ else $\max(f(l, false), f(r, false))$
l or r	True branch: if $f(l, true) \leq 0$ and $f(r, true) \leq 0$ then $f(l, true) + f(r, true)$ else $\max(f(l, true), f(r, true))$ False branch: if $f(l, false) \leq 0$ and $f(r, false) \leq 0$ then $f(l, false) + f(r, false)$ else $\min(f(l, false), f(r, false))$
not e	True branch: $f(e, false)$ False branch: $f(e, true)$

if the value of the branch function is positive, the desired branch is exercised; and (3) if the value of the branch function is increased but remains less than or equal to 0, the search that caused this change is considered successful.

Ferguson and Korel did not consider logical operators when defining branch functions. However, logical operators are common in spreadsheets, so it is necessary to handle them. To accomplish this we defined the branch functions shown in Table II. The purpose of these functions is to allow other branch functions to be combined in a meaningful way.

After calculating the branch function for a break point, the search procedure seeks a set of inputs that satisfies that branch function *without violating a constraint in the constraint path that is already satisfied*. This search involves a constrained linear search over inputs in *InputCells*, in which, following the procedure used by Ferguson and Korel [1996], a sequence of "exploratory" and "pattern" moves are applied over time (for efficiency, the search considers only those input cells in *InputCells* that could affect the target break point). Exploratory moves attempt to determine a direction of search on an input by incrementing or decrementing the input and seeing whether the value of the branch function improves, testing relevant inputs in turn until a candidate is found. Pattern moves continue in the same direction as a successful exploratory move, incrementing or decrementing values of a candidate input (by increasing or decreasing deltas), and seeing whether the value of the branch function

improves. If any move causes the value of the branch function to become positive, the break point has been covered, and the search terminates.

In the example that we have been using, the branch function for the break point at predicate node 23 is "if UnitsError $-1 = 0$ then 1 else $-|$UnitsError $-1|$." The input cells that could affect this branch function are ClipUO and PenUO. Assume that the search procedure first considers the cell PenUO, and that the input cells have initial values as shown in Figure 4. In this case, the branch function evaluates to -1 (line 7). First, the search procedure performs exploratory moves to try to determine in which direction the input should change (lines 9–19). PenUO is incremented and the branch function is recalculated, but still equals -1. Therefore, there is no improvement and the other direction is considered. However, when decremented the branch function still shows no improvement. Next, the algorithm considers input cell ClipUO, with similar results. At this point, the search procedure returns and node 23 is declared a problem break point.

Since node 23 is a problem break point, Chaining next considers the du-association (29, (23,T)), which has node 28 as its break point. The branch function for this break point is "if PenUO $-0 \leq 0$ and ClipUO $-0 \leq 0$ then 0$-$ PenUO $+0-$ ClipUO else 1." With the initial inputs from Figure 4, this branch function evaluates to -120. Once again, assume ClipUO is considered first. Its value is incremented, and the branch function now evaluates to -121, which is further from the goal. Then the input is decremented, and the branch function evaluates to -119, an improvement.

Now, the search procedure begins performing pattern moves in the negative direction on PenUO (lines 20–37). At each step, it doubles the size of the step until the value of the branch function no longer improves or some other constraint is satisfied. Thus, the next step is to change ClipUO to 17 (a step of -2), with the branch function evaluating to -117. Next, the algorithm makes a step of -4 to 13, with result -113, a step of -8 to 5 with result -105, and finally, a step of -16 to -11 with result 1, success. At this point, execution at node 28 is taking the true branch, and (29, (23,T)) is being exercised. This is exactly what needed to occur for the originally selected du-association of (24, (33,T)) to be exercised.

3.2.3 *Comparison of Approaches.* The differences between our rendering of the Chaining technique and Ferguson and Korel's original rendering primarily involve situations in which we have been able to simplify Ferguson and Korel's approach, due to characteristics of the spreadsheet evaluation model. These simplifications improve the efficiency of the approach—a critical matter in spreadsheets, which feature rapid response. We summarize the primary differences here. To facilitate the presentation, we refer to Ferguson and Korel's technique as CF-Chaining, reflecting its use of control flow information.

—Given a problem break point, CF-Chaining requires data flow analysis to compute definitions of variables that may affect flow of control in that problem break point. Such computation can be expensive, particularly in programs for which aliasing must also be considered [Pande et al. 1994]. In contrast, Chaining takes advantage of data dependence information computed

incrementally by the spreadsheet engine during its normal operation. Such computation adds $O(1)$ cost to the operations already performed by that engine [Rothermel et al. 2001] to update the visual display.

—CF-Chaining builds *event sequences* that encode lists of nodes that may influence a test's ability to execute a goal node; these sequences are constructed as problem break points are encountered, and used to guide attempts to solve constraints that affect the reachability of problem break points. For the reasons just stated, Chaining is able to take a more direct approach, as the information encoded in event sequences is already available in the CRG (which also is computed by the spreadsheet engine, at $O(1)$ cost above the operations it already must perform [Rothermel et al. 2001]).

—CF-Chaining's algorithm is complicated by the presence of loops which create *semicritical* branches that may or may not prevent reaching a problem break point. CF-Chaining uses a branch classification scheme to differentiate semicritical branches, *critical* branches that necessarily affect reachability, and *nonessential* branches that require no special processing. Since the spreadsheets that we consider do not contain loops, Chaining does not need this branch classification; instead, it can build constraint paths directly from branches occurring in the defining and using cells.

—The presence of loops in imperative programs also requires CF-Chaining to impose a limit on depth of chaining that Chaining does not need to impose.

—CF-Chaining does not, as presented, include a technique for handling logical operators; the technique we present here for use with Chaining could be adapted to function with CF-Chaining.

3.3 Supplying and Using Range Information

The random test generation technique requires ranges within which to randomly select input values, and the chaining technique needs to know the edge of its search space. One scenario in which the techniques can operate is one in which no specific information on expected cell ranges is available. In this scenario, the techniques may need to consider all possible cell values within the default range of each cell's data type.

A second scenario in which test generation techniques can operate is the scenario in which, via a user's help or the application of various algorithms [Burnett et al. 2003], more precise knowledge of expected cell ranges is obtained. For example, there has been research on eliciting and deducing assertions that can specify a cell's expected values [Burnett et al. 2003; Ernst et al. 2001; Raz et al. 2002]. Using such range information, both techniques can limit their search space to the specified ranges and generate test inputs within these ranges.

It seems likely that the use of range information might improve the efficiency and effectiveness of test generation techniques, at the cost, to the system or user, of obtaining that range information. To facilitate investigation of this issue, we implemented our test generation techniques in a manner that supports each of the foregoing scenarios. Our empirical studies compare the techniques with and without range information.

Table III.

Data Type	Input Cells	Referenced Input Cells	Formula Cells
Blank	N/A	N/A	110
Error	25	0	238
Boolean	29	8	24
Date	1320	66	136
Real	2257	815	1139
Integer	3352	1486	1730
String	4481	448	454

Number of spreadsheets that contain input cells (cells containing only constants), referenced input cells (input cells that are referenced in some formula in the spreadsheet), and formula cells (cells that are not input cells and that compute values) of various data types. Data is drawn on a sample of 4498 spreadsheets obtained primarily through Google searches on keywords common to spreadsheets, plus 85 spreadsheets that had been used in prior spreadsheet studies reported in the literature.

3.4 Supporting Other Data Types

Our discussion of test generation thus far, and our initial test generation prototype, have focused on spreadsheets that contain integer input cells, and predicates with numeric relational operators combined with the standard logical operators `and`, `or`, and `not`. However, the results of a survey of 4498 spreadsheets [Fisher and Rothermel 2005], summarized in Table III, show that although integer types are very important in spreadsheets, it is important to consider how to support other datatypes. To provide support for these types, it is necessary to consider their uses in both predicate expressions (for `Chaining`) and input cells (for both `Chaining` and `Random`).

First, we consider supporting additional datatypes in predicate statements. This requires considering all relational operators and predicates that work with that datatype and defining suitable branch functions. A minimal level of support for this can be achieved by defining a result of `true` from the relational operator as "1" and `false` as "-1." Unfortunately, these definitions do not allow `Chaining` to directly solve these branch functions in most cases, as small changes in the input are not likely to solve the break point, and there is no possibility of improving the value of the branch function without solving it. However, in practice it isn't necessary for `Chaining` to directly address these cases; rather, `Chaining` can be invoked to cover du-associations downstream from the predicates, in which case it is still possible to chain back and arrive at a solution. An example of this involves the use of strings to indicate errors. The spreadsheet `Budget`, in Figure 3, uses the values 1 and 0 in the cell `UnitsError` to indicate out-of-range values in input cells. These could be changed to more descriptive strings, with appropriate adjustments to `TotalCost`, and `Chaining` would be just as effective at covering the du-associations that depend on `UnitsError`.

A more difficult problem is providing support for noninteger inputs. Although the results in Table III show that integers are by far the most frequently referenced input type, other types still need to be considered. For some, such as Boolean or error values, the types can be considered as simple enumeration types with a relatively small range of values, and trying all values within the type is reasonable within the search step of the `Chaining` algorithm

(or sampling from all possible values with the Random algorithm). For other types, such as reals and strings, more consideration is needed.

The most important requirement within the scope of Chaining is a minimum step size. For integers, the minimum step size was set at 1, the minimum difference between any two integers. This works because the difference between any value in integer space and its neighbors is constant throughout the space. Of course, this is not true of all domains. For example, in floating point space, the difference between consecutive values is much smaller for values close to 0 than it is for large magnitude values. It would theoretically be possible to use the nearest neighbors to the current value as a minimum step size, but experimentation will be needed to determine whether such an approach can succeed in affecting branch functions after considering round-off error and similar anomalies. For Random, an additional requirement of minimum and maximum values for the datatype is also necessary. For integers we use user-supplied ranges, or in the absence of these ranges, the minimum and maximum values of a particular representation of integers used in our system. For other datatypes, defining the range of values in a similar fashion may be problematic.

4. EMPIRICAL STUDIES

Our test generation methodology is intended to help spreadsheet programmers achieve du-adequacy in testing, and this du-adequacy is communicated to the user with devices such as cell border colors. Determining whether this methodology achieves this goal will ultimately require user studies; however, such studies are expensive, and before undertaking them there are several more fundamental questions about the methodology that need to be answered. In particular, we need to determine whether the methodology can successfully generate inputs to exercise a large percentage of the du-associations in spreadsheets, and whether it can do so sufficiently efficiently. If the methodology is not sufficiently effective and efficient, there is no reason to pursue expensive studies involving human subjects. Moreover, if the methodology does show promise, the question of what flavor(s) of the methodology to investigate on humans (random or goal-oriented, with or without range information, spreadsheet, cell, or du-association level) must also be answered prior to conducting human subject studies.

For these reasons, in this work we address the following research questions, the first concerning effectiveness, and the latter two concerning efficiency:

*RQ*1. Can our test generation methodology generate test inputs that execute a large proportion of the feasible du-associations of interest?

*RQ*2. How long will a user have to wait for our methodology to generate a test input?

*RQ*3. How long will a user have to wait for our methodology to report that it cannot generate a test input?

In addressing these questions we also consider the differences between Random and Chaining, and whether provision of range information has any effect on results. Further, we investigate whether differences in results occur across the

Table IV. Data about Subject Spreadsheets

Spreadsheets	Cells	DU-assoc's	Feasible DU-assoc's	Expressions	Predicates
Budget	25	56	50	53	10
Digits	7	89	61	35	14
FitMachine	9	121	101	33	12
Grades	13	81	79	42	12
MBTI	48	784	780	248	100
MicroGen	6	31	28	16	5
NetPay	9	24	20	21	6
NewClock	14	57	49	39	10
RandomJury	29	261	183	93	32
Solution	6	28	26	18	6

three levels at which users can apply the methodology (spreadsheet, cell, and du-association).

To proceed with this investigation, we prototyped our test generation methodology, including both the `Random` and `Chaining` techniques, in Forms/3. Our prototypes allow test generation at the whole spreadsheet, selected cell, and selected du-association levels, with and without supplied range information.

4.1 Subjects

We used ten spreadsheets as subjects (see Table IV). These spreadsheets had previously been created by experienced Forms/3 users to perform a wide variety of tasks: `Digits` is a number to digits splitter, `Grades` translates quiz scores into letter grades, `FitMachine` and `MicroGen` are simulations, `NetPay` calculates an employee's income after deductions, `Budget` determines whether a proposed purchase is within a budget, `Solution` is a quadratic equation solver, `NewClock` is a graphical desktop clock, `RandomJury` determines statistically whether a panel of jury members was selected randomly, and `MBTI` implements a version of the Myers-Briggs type indicator (a personality test). Table IV provides data indicating the complexity of the spreadsheets considered, including the numbers of cells, du-associations, expressions, and predicates contained in each spreadsheet. Although these spreadsheets may seem small, Section 4.8 shows that a significant number of real spreadsheets are of similar complexity with respect to the effort they require of WYSIWYT and automatic test generation.

Our test generation prototype handles only integer type inputs; thus, in selecting subject spreadsheets we restricted our focus to those that utilize only integer data types. Since commercial spreadsheets contain infeasible du-associations, however, we made no attempt to limit the presence of these in our subject spreadsheets. We did, however, inspect the subject spreadsheets to determine which du-associations in those spreadsheets were feasible and which were infeasible.

4.2 Measures

Because our underlying testing system uses du-adequacy as a testing criterion and our goal is to support the use of this criterion, we measure a test generation

technique's *effectiveness* in terms of the percentage of feasible du-associations exercised by the test inputs it generates.

As a measure of a test generation technique's *efficiency* we choose *response time*: The amount of wall clock time required to generate a test input that exercises one or more du-associations, or to report that one cannot be generated. This represents the period of time a user might have to wait for a response, and in interactive spreadsheet systems, this is crucial.

We expect that effectiveness and response time will differ across du-associations, and that as the coverage of the feasible du-associations associated with a spreadsheet, selected cell, or selected du-association arrow nears 100%, the time required to generate a new useful test input will increase. Thus, we gather these metrics incrementally over the course of repeated attempts to generate test inputs for a spreadsheet, selected cell, or selected du-association arrow, automatically recording the new cumulative coverage reached whenever a new test input is generated and recording the final level of coverage achieved, continuing until all relevant du-associations have been exercised, or a long time limit has been reached (we discuss issues involving time limits and experiment procedures further in the next section).

4.3 Methodological Considerations

When employed by an end user under our methodology, our test generation techniques generate one test at a time. However, the user may continue to invoke a technique to generate additional test inputs. We expect that within this process, as the coverage of the feasible du-associations in a spreadsheet nears 100%, the remaining du-associations will be more difficult to cover, and the time required to generate a new useful test input will increase. We wish to consider differences in response time across this process. Further, it is only through repeated application of a technique that we can observe the technique's overall effectiveness. Thus, in our experimentation we simulate the process of a user repeatedly invoking "Help Me Test" by applying our test generation techniques repeatedly to a spreadsheet, selected cell, or selected du-association arrow in a controlled fashion. To achieve this, we use automated scripts that invoke our techniques and gather the required measurements. This approach raises several issues, as follows.

4.3.1 *Automatic Correctness Indications.* The testing procedure under WYSIWYT is divided into two steps: Finding a test input that executes one or more unexercised du-associations in the spreadsheet, and determining and indicating the correctness of appropriate cell values. In these studies, because we are interested only in the test input generation step and do not have users performing correctness indications, our scripts automatically perform these indications on all relevant validatable output cells, causing all du-associations contributing to the production of values in these cells to be marked "exercised." This approach simulates the effects of users' correctness indications under the assumption that, following the application of a generated test input, the user indicates the correctness of all cells in the area of interest. We do not measure correctness indication time as part of our response time measurement.

4.3.2 *Time Limit.* To simulate a user's continued calls to test generation techniques during incremental testing, our scripts repeatedly apply the techniques to the subject spreadsheet after each (automatic) correctness indication. In practice, a user or internal timer might stop a technique if it ran "too long." In this study, however, we wish to examine effectiveness and response time more generally and discover what sort of internal time limits might be appropriate. Thus, our scripts must provide sufficiently, long times in which our techniques can attempt to generate test inputs.

Obviously, our techniques would halt if 100% du-adequacy were achieved; however, since each subject spreadsheet contains infeasible du-associations, and our generators are not informed as to which du-associations are executable, we cannot depend on this condition to occur. Moreover, even when applied to feasible du-associations, we do not know whether our test generation techniques can successfully generate test inputs for those du-associations. Thus, in our experiments we use a timer with a long time limit to specify how long the techniques will continue to attempt to generate test inputs.

To determine what time limits to use, we performed several trial runs at the spreadsheet, cell, and du-levels with extremely long limits (e.g., several hours at the spreadsheet level). We then determined the maximum amount of time in these runs after which no additional test inputs were found, and set our limits at twice this amount of time. Obviously, we cannot guarantee that longer time limits would not allow the techniques to exercise additional du-associations. However, our limits certainly exceed the time that users are likely to wait for test generation to succeed, and thus, for practical purposes are sufficient.

4.3.3 *Feasible and Infeasible Du-Associations.* For the purpose of *measuring* effectiveness, we consider only coverage of feasible du-associations: this lets us make effectiveness comparisons between subjects containing differing percentages of infeasible du-associations. We can take this approach because we already know, through inspection, the infeasible du-associations for each spreadsheet. In practice, however, our techniques would be applied to spreadsheets containing both feasible and infeasible du-associations, and might spend time attempting to generate inputs for both. Thus, when we *apply* our techniques we do not distinguish between feasible and infeasible du-associations; this lets us obtain fair response time measurements.

4.3.4 *Range Information.* Our research questions include those about the effects of input ranges, and our experiments investigate the use of techniques with and without range information. For cases in which no range information is provided, we used the default range for integers (-536870912 to $+536870911$) on our system. We determined that this range was large enough to provide inputs that execute each feasible du-association in each of our subject spreadsheets.

To investigate the use of ranges, we needed to provide ranges such as those that might be provided by a user of the system. To obtain such range information for all input cells in our subject spreadsheets, we carefully examined the spreadsheets, considering their specifications and their formulas, and created

an original range for each input cell that seemed appropriate based on this information. To force consideration of input values outside of expected ranges, which may also be of interest in testing, we then expanded these initial ranges by 25% in both directions (in practice, such an expansion could be accomplished by the user, or by the test generation mechanism itself).

4.3.5 *Initial Values.* In general, another consideration that might affect the effectiveness and response time of techniques is the initial values present in cells when a test generator is invoked. `Random` randomly generates input values until it finds useful ones, and thus, is independent of initial values. `Chaining`, however, starts from the current values of input cells and searches the input space under the guidance of branch functions until it finds a solution, so initial values can affect its performance.

To control for the effects of initial values, we performed multiple runs using different initial cell values on each spreadsheet. Further, to control for effects that might bias comparisons of the techniques, we used the same sets of initial values for each technique. All initial values thus fell within ranges.

4.4 Experiment Design

Our experiments were performed at all three abstraction levels: spreadsheet, selected cell, and selected du-association level. In other words, we attempted to generate test inputs that exercise all du-associations in a spreadsheet, all du-associations associated with (ending at) a cell, and all du-associations associated with a du-association arrow.

In all three experiments, three independent variables were manipulated:

—the ten spreadsheets;
—the test generation technique;
—the use of range information.

We measured two dependent variables:

—effectiveness;
—response time.

All runs were conducted, and all timing data collected, on Sun Microsystems Ultra 10s with 128 MB of memory, with all output written to local disk storage. During the runs, our processes were the only user processes active on the machines.

4.4.1 *Spreadsheet-Level Experiment Design Specifics.* For each subject spreadsheet we applied each of our two test generation techniques, with and without range information, starting from 35 sets of initial values (i.e., initial assignments of values to all input cells). On each run, we recorded the times at which test generation began, and the times at which untested du-associations were exercised or at which test generation failed; these measurements provided the values for our dependent variables. These runs yielded 1400 sets of effectiveness and efficiency measurements for our analysis.

4.4.2 *Cell-Level Experiment Design Specifics.* For each spreadsheet, we ran a number of trials equal to three times the number of noninput cells in that spreadsheet. As in the spreadsheet level experiment, we randomly assigned initial values. We then randomly chose a cell that was not fully tested by these initial values. The two test generation techniques were then applied to the selected cell under the given input assignment, with and without range information (on each run the system was reset and initial values restored, as described above.) On each run, we recorded the times at which test generation began, and the times at which untested du-associations were exercised or at which test generation failed; these measurements provided the values for our dependent variables. This process yielded (number-of-testable-cells-across-all-spreadsheets) $\times 3 \times 2 \times 2$ sets of effectiveness and efficiency measurements for our analysis.

4.4.3 *Du-Association-Arrow-Level Experiment Design Specifics.* For each spreadsheet, we ran a number of trials equal to the number of feasible du-associations in that spreadsheet starting from randomly assigned initial inputs. We then randomly chose a feasible du-association arrow that was not fully tested by these initial values. The two test generation techniques were then applied to the selected du-association arrow, with and without range information, under the given input value assignment. On each run, we recorded the times at which test generation began, and the times at which untested du-associations were exercised or at which test generation failed; these measurements provided the values for our dependent variables. This setup yielded (number-of-feasible-du-associations-across-all-spreadsheets) $\times 2 \times 2$ sets of effectiveness and efficiency measurements for our analysis.

4.5 Threats to Validity

All experiments have limitations (threats to validity) that must be considered when assessing their results. The primary threats to validity for this experiment are external, involving subject and process representativeness, affecting the ability of our results to generalize. Our subject spreadsheets are of small and medium size, with input cells only of integer type. Commercial spreadsheets with different characteristics may be subject to different cost-effectiveness trade-offs. Our experiment uses scripts that automatically validate all relevant output cells; in practice, a user may validate some or none of these cells. Our range values were created by examining the subject spreadsheets, but may not represent the ranges that would be assigned by users in practice. The initial values we assigned to cells fall within these ranges, but might not represent initial values that would typically be present when a user requested help in test generation. Threats such as these can be addressed only through additional studies using other spreadsheets, and studies involving actual users.

Threats to internal validity involve factors that may affect dependent variables without the researcher's knowledge. We considered and took steps to limit several such factors. First, test generation techniques may be affected by differences in spreadsheets and formulas; to limit this threat, our experiments utilized a range of spreadsheets that perform a variety of tasks. Second, initial

input cell values can affect the success and speed of Chaining; we limit this threat by applying techniques repeatedly (35 times per spreadsheet) using different initial values. Third, timings may be influenced by external factors such as system load and differences among machines; to control for this, we ran our experiments on a single machine on which our processes were the only user processes present. Finally, to support fair timing comparisons of our techniques, our implementations of techniques shared code wherever possible, differing only where required by the underlying algorithms.

Threats to construct validity occur when measurements do not adequately capture the concepts they are supposed to measure. Degree of coverage is not the only possible measure of effectiveness of a test generation technique; fault detection ability and size of the generated test suite may also be factors. Moreover, certain techniques may generate output values that are easier for users to validate than others, affecting both effectiveness and efficiency.

4.6 Data and Analysis

We now present and analyze the data from our experiments, considering each research question in turn, using both descriptive and inferential statistics.[2] Section 4.7 discusses implications of these results.

4.6.1 *Research Question 1: Ability to Generate Test Inputs.* We turn first to our first research question: whether our test generation methodology can generate test inputs that execute a large proportion of the feasible du-associations of interest. In our analysis we consider two different views of effectiveness. First, we consider the ability of our techniques to generate test inputs to cover all the feasible du-associations; we refer to this as their *ultimate effectiveness*. Second, we consider the range of coverage levels achieved across runs. Each of these analyses can best be accomplished on results collected on whole spreadsheets, and thus, we address them using data from just the spreadsheet-level runs.

4.6.1.1 *Ultimate Effectiveness.* Ultimate effectiveness is simply a percentage measure of how many feasible du-associations in a spreadsheet can be exercised by a method. Table V lists, for each subject spreadsheet, the ultimate effectiveness of Random and Chaining, with and without range information, averaged across 35 runs.

As the table shows, Chaining without range information achieved over 99% average ultimate effectiveness on all but two of the spreadsheets (FitMachine and RandomJury). On these two spreadsheets, the technique achieved average ultimate effectiveness over 97% and 94%, respectively.

We had expected that the use of range information would improve the effectiveness of Chaining. A comparison of the values in the two rightmost columns in Table V, however, suggests that there was little difference in average ultimate effectiveness between Chaining with and without range information. In fact, on FitMachine and RandomJury, the two cases in which there was the greatest potential for improvement, addition of range information actually decreased

[2]All statistical data presented in this section was obtained using StatView 5.0.

Table V. Ultimate Effectiveness of Techniques per Spreadsheet (Averaged over 35 Runs)

	Random without Ranges	Random with Ranges	Chaining without Ranges	Chaining with Ranges
Budget	96.6%	100.0%	100.0%	100.0%
Digits	59.4%	97.9%	100.0%	100.0%
FitMachine	50.5%	50.5%	97.9%	97.9%
Grades	67.1%	99.8%	99.7%	99.9%
MBTI	25.6%	100.0%	99.9%	99.6%
MicroGen	71.4%	99.2%	100.0%	100.0%
NetPay	40.0%	100.0%	100.0%	100.0%
NewClock	57.1%	100.0%	99.0%	99.4%
RandomJury	78.8%	83.2%	94.3%	92.7%
Solution	57.7%	78.8%	100.0%	100.0%

average ultimate effectiveness, though by less than 2%. To determine whether the differences in average ultimate effectiveness between Chaining with and without range information were statistically significant, we used paired t-tests on pairs of effectiveness values per technique per spreadsheet. The differences between the techniques were statistically significant only for MBTI ($\alpha < .05$).

Random without range information behaved much differently than Chaining, in only one case achieving an average ultimate effectiveness greater than 90% (Budget), and in six of ten cases achieving an average ultimate effectiveness less than 60%. Ultimate effectiveness also varied widely for this technique, ranging from 25.6% to 96.6%. On all ten spreadsheets, the average ultimate effectiveness of Random without ranges was less than that of Chaining without ranges; differences between the techniques ranged from 3.4% to 74.2% across spreadsheets (average overall difference 38%). Paired t-tests showed that the effectiveness differences between Random without ranges and Chaining without ranges were all statistically significant ($\alpha < .05$).

In contrast to the results observed for Chaining, addition of range information to Random did affect its performance, in all but one case increasing average ultimate effectiveness, and in seven of ten cases increasing it by more than 20%. Paired t-tests showed that all increases were statistically significant; effectiveness remained unchanged only on FitMachine.

Addition of range information to Random also helped its performance in comparison to Chaining. On two spreadsheets, MBTI and NewClock, Random with range information achieved greater average ultimate effectiveness than Chaining with range information. However, this difference, though statistically significant, was less than 1%. On five spreadsheets (Digits, FitMachine, MicroGen, RandomJury, and Solution), on the other hand, Chaining with range information resulted in statistically greater average ultimate effectiveness than Random with range information, and in two of these cases the difference exceeded 20% (on Grades, NetPay, and Budget, differences were not statistically significant).

4.6.1.2 *Variance in Coverage Levels.* Our second view of effectiveness considers the variance in final coverage levels reached for each technique on each spreadsheet, in terms of the minimum and maximum levels achieved across the

Table VI. Variance in Coverage Levels Achieved per Technique per Spreadsheet across 35 Runs

	Random without Ranges		Random with Ranges		Chaining without Ranges		Chaining with Ranges	
	min	max	min	max	min	max	min	max
Budget	96.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Digits	21.3	75.4	95.1	100.0	100.0	100.0	100.0	100.0
FitMachine	50.5	50.5	50.5	50.5	97.0	98.0	97.0	98.0
Grades	67.1	67.1	98.7	100.0	89.9	100.0	96.2	100.0
MBTI	25.6	25.6	100.0	100.0	99.3	100.0	98.9	100.0
MicroGen	71.4	71.4	92.9	100.0	100.0	100.0	100.0	100.0
NetPay	40.0	40.0	100.0	100.0	100.0	100.0	100.0	100.0
NewClock	57.1	57.1	100.0	100.0	95.9	100.0	95.9	100.0
RandomJury	78.7	79.8	79.8	86.9	72.7	97.3	78.7	97.8
Solution	57.7	57.7	57.7	80.8	100.0	100.0	100.0	100.0

35 runs of that technique on that spreadsheet. This variance is an indicator of the degree of consistency that can be expected in technique effectiveness across applications.

Table VI shows the minimum and maximum coverage reached by each technique on each spreadsheet, with and without range information, across the 35 different runs of that technique at the spreadsheet-level. For example, the first two columns of the table indicate that when the Random technique without ranges was run on the `Digits` spreadsheet, the lowest coverage achieved over the 35 different executions was 21.3%, and the highest was 75.4%. The same technique run on the `Grades` spreadsheet produced equivalent minimum and maximum coverages of 67.1%; in other words, all 35 runs reached that coverage level. As the table shows, in most cases the variance was small (on five spreadsheets it was less than 5% for all techniques).

We had expected to see the largest variance for `Random` without ranges, and to a lesser extent, for `Random` with ranges. Instead, `Random` without ranges exhibited no variance in coverage achieved on seven of ten spreadsheets, and less than 4% variance on two others, with a wide variance only on `Digits`. `Random` with ranges, while more consistent in coverage achieved than we expected, did have variance equal to or larger than that for `Random` without ranges on all spreadsheets except `Digits` and `Budget`.

We also expected that `Chaining` would be more consistent than `Random`. This expectation was not met, however, on five of the spreadsheets (`FitMachine`, `Grades`, `MBTI`, `NewClock`, and `RandomJury`), either with or without range information. However, the degree of variance in coverage levels using `Chaining` on these five spreadsheets was usually small, exceeding 5% only on `Grades` and `RandomJury` without ranges, and only on `RandomJury` with range information.

We also expected range information to lower variance in coverage for `Chaining`, and this did occur on all spreadsheets except for `FitMachine`, which had a coverage range of .5% more with ranges than without.

4.6.2 *Research Question 2: Response Time on Success.* We turn next to our second question: How long should a user expect to wait for a response after clicking "Help Me Test" in the case in which test generation is successful?" This

question distinguishes *successful* test generation attempts from unsuccessful ones; we do this because response times for Random on unsuccessful attempts are dictated only by an internally set constant, and including unsuccessful measures for Random in a response time comparison artificially skews the data (we consider unsuccessful attempts for Chaining in Section 4.6.3).

We consider three views of the data. First, we consider response times independently of other factors. Second, we consider response time as cumulative coverage increases. These first two views are taken relative only to spreadsheet-level data; our third view considers differences in response times among levels of technique application (spreadsheet, cell, and du-association arrow).

4.6.2.1 *Successful Response Times, Spreadsheet Level.* Figure 9 presents boxplots showing all of the response data gathered in the spreadsheet-level experiment for Random and Chaining with and without ranges, per spreadsheet.[3] For ease of reference, Table VII presents the median response times shown in the plots.

As the graphs and table show, the median response times for Random with and without ranges was less than 10 seconds on six of the ten spreadsheets. Chaining with and without ranges exhibited similar performance on only two of the spreadsheets (NetPay and MicroGen), although it was close to 10 seconds on several others. All of the median response times, however, were less than 40 seconds, except those for three of the four techniques applied to RandomJury where times ranged from 80 to 182 seconds.

Comparing Chaining to Random, unpaired t-tests showed that response times for Chaining without ranges were statistically significantly larger than response times for Random without ranges on all but one spreadsheets ($\alpha < .05$). With range information added, statistically significant differences between Chaining and Random continued to be observed for all spreadsheets except Digits, Grades, MBTI, and MicroGen.

We expected that range information would improve response times for both Chaining and Random; however, this was not generally the case. For Chaining there were only small differences in median response times with and without ranges, usually less than 1 second. Considering the distribution of response times shown in the boxplots, distributions for Chaining both with and without ranges appear similar in all but one case (Digits), and unpaired t-tests found statistically significant differences in only this case.

Differences between Random with and without ranges, in contrast, were more mixed, with range information helping significantly on RandomJury, the absence of range information helping significantly on Digits, Grades, MBTI, MicroGen, and NetPay, and no significant difference observed for Budget, FitMachine,

[3]A boxplot is a standard statistical device for representing data distributions; boxplots have an advantage over simple averages in that they better depict the variance among elements of the data set. In the boxplots presented in this article, each data set's distribution is represented by a box. The box's width spans the central 50% of the data, and its ends mark the upper and lower quartiles. The vertical line partitioning the box represents the median element in the data set. The horizontal lines attached to the ends of the box indicate the tails of the distribution. Data elements considered outliers are depicted as circles.

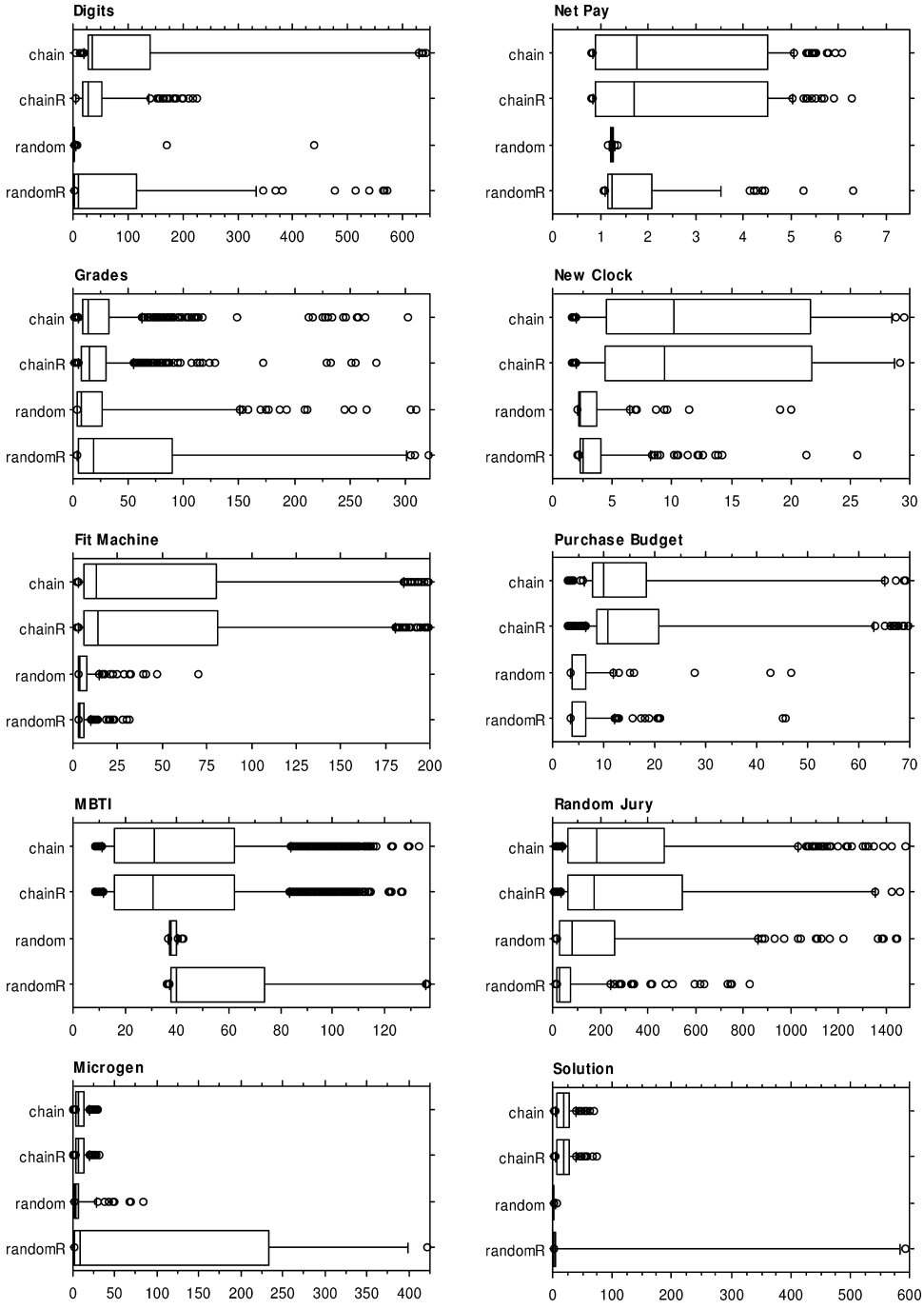

Fig. 9. Response times (seconds) for experiments at the spreadsheet level. Technique abbreviations are: "chain" = Chaining without ranges, "chainR" = Chaining with ranges, "random" = Random without ranges, "randomR" = Random with ranges.

Table VII. Median Response Times per Spreadsheet, Spreadsheet-Level Experiments

	Random without Ranges	Random with Ranges	Chaining without Ranges	Chaining with Ranges
Budget	3.8	3.9	9.9	10.6
Digits	2.2	10.1	34.5	28.4
FitMachine	3.8	3.9	13.5	14.3
Grades	7.7	18.6	14.2	14.4
MBTI	37.5	40.0	31.2	30.7
MicroGen	2.7	8.1	6.4	6.5
NetPay	1.2	1.2	1.7	1.7
NewClock	2.3	2.5	10.4	9.3
RandomJury	80.2	28.7	182.3	173.2
Solution	1.3	1.4	18.9	17.9

`NewClock`, and `Solution`. We suspect that the larger number of cases favoring absence of range information is due, however, to the low levels of coverage actually reached by `Random` without ranges. For those cases in which `Random` without ranges succeeds, it typically does so quickly, whereas range information causes additional, longer-response-time successes to be achieved.

4.6.2.2 *Successful Response Times versus Coverage Reached, Spreadsheet-Level.* As the preceding observation reveals, the two views of the data just considered do not account for differences in the numbers of test cases generated by different techniques, or for changes in response times as coverage increases. Higher response times may be expected and acceptable as full coverage is neared in a spreadsheet.

Figure 10 plots response time against coverage level reached for `Random` and `Chaining` with and without range information, per spreadsheet. Each plot contains four lines, one per technique and range information combination, distinguished as indicated by the legend. Each point plotted represents the mean response time exhibited by the given technique and range information combination on all test cases that, when generated, resulted in a level of total coverage of the spreadsheet in the 10% range of coverage enclosing that point (as labeled on the x axis). The lines thus show the growth in average response times across discrete 10% levels of coverage reached. Note that, since each experiment begins with a certain number of du-associations coverable prior to test generation, these lines do not begin at the $x = 0$ point, but rather at the point in which test generation begins to add coverage. Note also that in a few cases some overplotting occurs, with lines falling on top of one another and not distinguishable. A further limitation of the coverage graphs lies in the fact that the number of observations used to produce the medians are not shown; thus, if one technique reaches 90% coverage just once across all experiments, its one response time plotted in the (90–100] interval may represent an outlier. Keeping these limitations in mind, however, the graphs still facilitate interpretation.

One observation of interest in the plots is how the behaviors of `Chaining` with and without range information closely parallel each other throughout all graphs other than the one for `Digits`. This shows that, even considered over the course

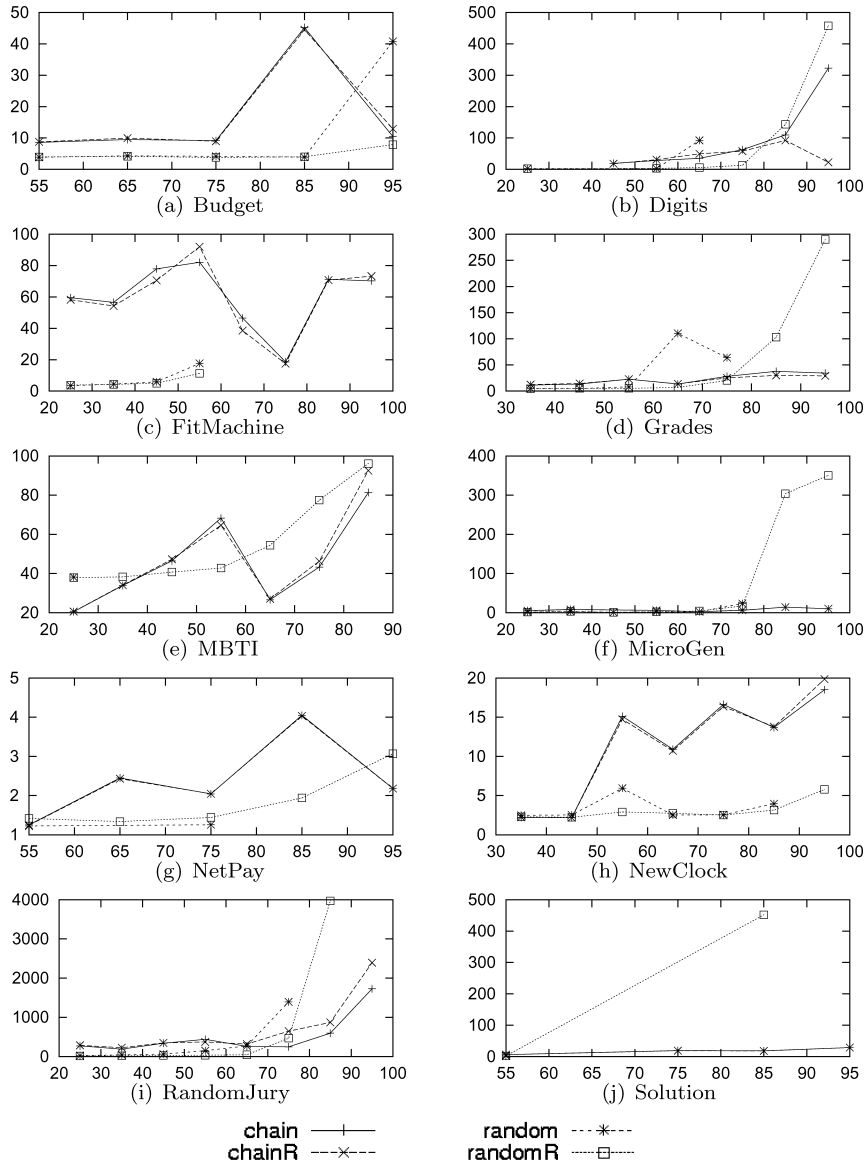

Fig. 10. Response times (seconds) as coverage increases for experiments at the spreadsheet level. Technique abbreviations are: "chain" = Chaining without ranges, "chainR" = Chaining with ranges, "random" = Random without ranges, "randomR" = Random with ranges.

of a testing session in which coverage increases, provision of range information had only slight effects on response time. The same observation holds for `Random` with and without ranges, except that `Random` without ranges typically achieved no coverage at higher coverage levels. This supports our earlier conjecture that the cases in which range information caused `Random` to attain higher median response times were due to its increased chances of success (although requiring

more work) at covering du-associations not coverable in the absence of range information.

Comparing Chaining with Random, it appears that Chaining had a smaller response time trend as coverage increased on six of the spreadsheet-level subjects (Digits, Grades, MBTI, MicroGen, RandomJury, and Solution). The remaining four spreadsheets showed varying trends. Three (FitMachine, NetPay, NewClock) of the Random without range plots have no representation at higher coverage levels. Two (NetPay and Budget) show Chaining trending toward longer response times until the (70% . . . 80%] range, at which point the Chaining response times improved to nearly equal to or better than the Random response times. Random reached similar coverage levels and maintained lower response time trends than Chaining on only two spreadsheets (Budget and NewClock), but only with range information.

4.6.2.3 *Successful Response Times—Differences among Levels.* Our final view of response times concerns whether those times vary across the three levels of test generation: spreadsheet, individual cell, and individual du-association arrow. Figures 11 and 12, similar to Figure 9, present boxplots showing all response data gathered in the cell and du level experiments, respectively. For ease of reference, Tables VIII and IX list just the median times shown on the graphs.

In terms of relationships between techniques, results at the cell and du-association arrow levels were fundamentally similar to those observed at the spreadsheet level. Response times for Chaining were again statistically significantly greater than those for Random in most cases, with or without ranges (the exceptions being with ranges, NetPay at the cell level, and without ranges, NetPay and MBTI at cell and du-association arrow levels). Range information did not significantly improve response times of Chaining or Random on any spreadsheet at either level, although it did significantly detract from response times in some cases on Random (at the cell level: Digits, Grades, and MBTI; at the du-association arrow level: these three and MBTI).

Comparisons of results across levels are somewhat mixed. In terms of median response times, Random (with and without range information) performed more quickly at the cell and du-association levels than at the spreadsheet level in most cases, most exceptions being those in which response times were already low at the spreadsheet level. Median response times for Chaining, on the other hand, were often larger at the cell and du-association levels than at the spreadsheet level, a notable exception being RandomJury, on which large median response times were reduced by between one-half and two-thirds at the lower levels of application.

A more substantial and pervasive difference between levels of application can be seen in the boxplots. In most cases, the range of response times exhibited by the Chaining technique (with and without ranges) was smaller at the cell and du-association arrow levels than at the spreadsheet levels, regardless of whether median response times were higher or lower at the latter level. This is particularly evident in the reduced size of the second and third quartiles in

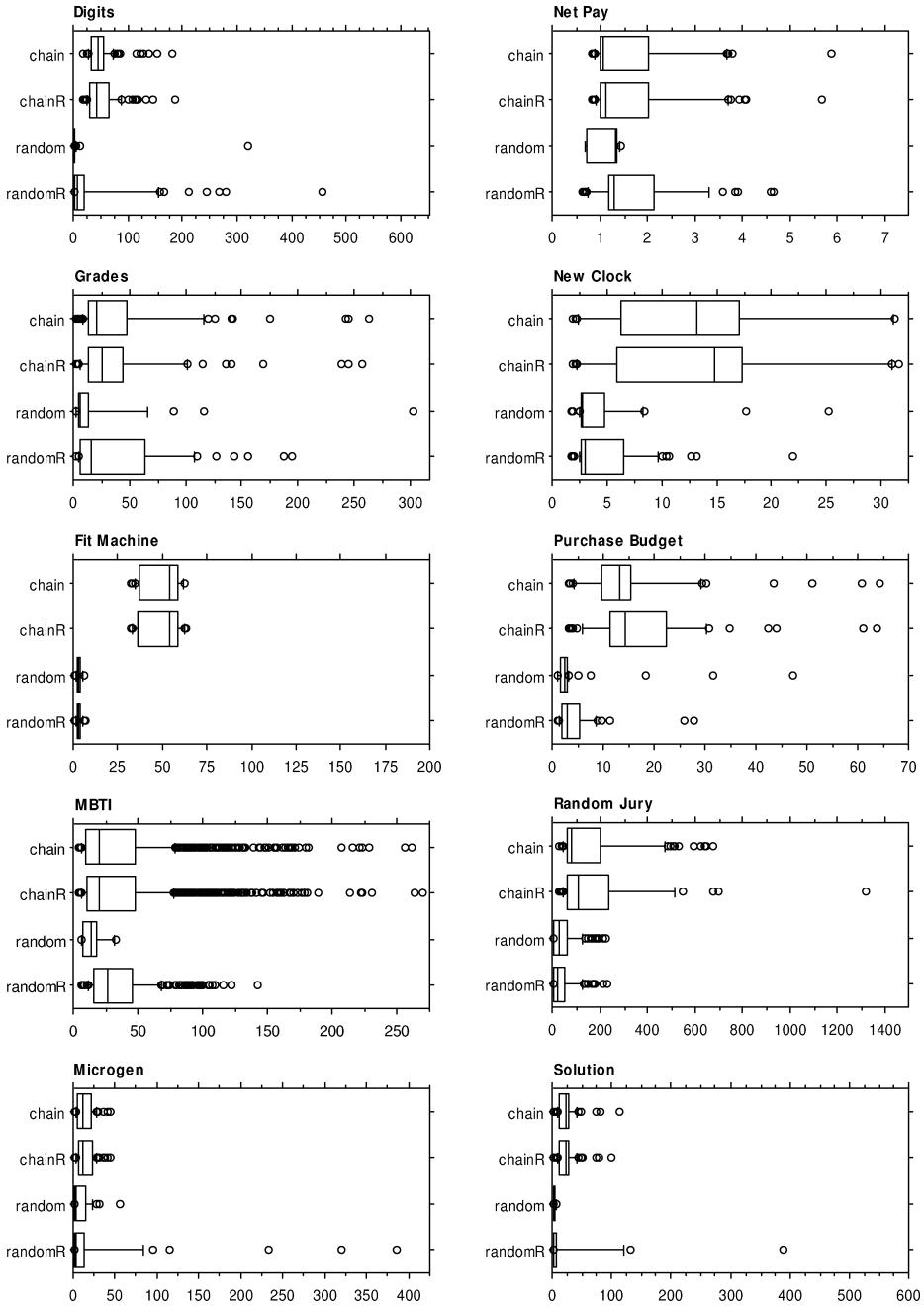

Fig. 11. Response times (seconds) for experiments at the cell level. Technique abbreviations are: "chain" = Chaining without ranges, "chainR" = Chaining with ranges, "random" = Random without ranges, "randomR" = Random with ranges.

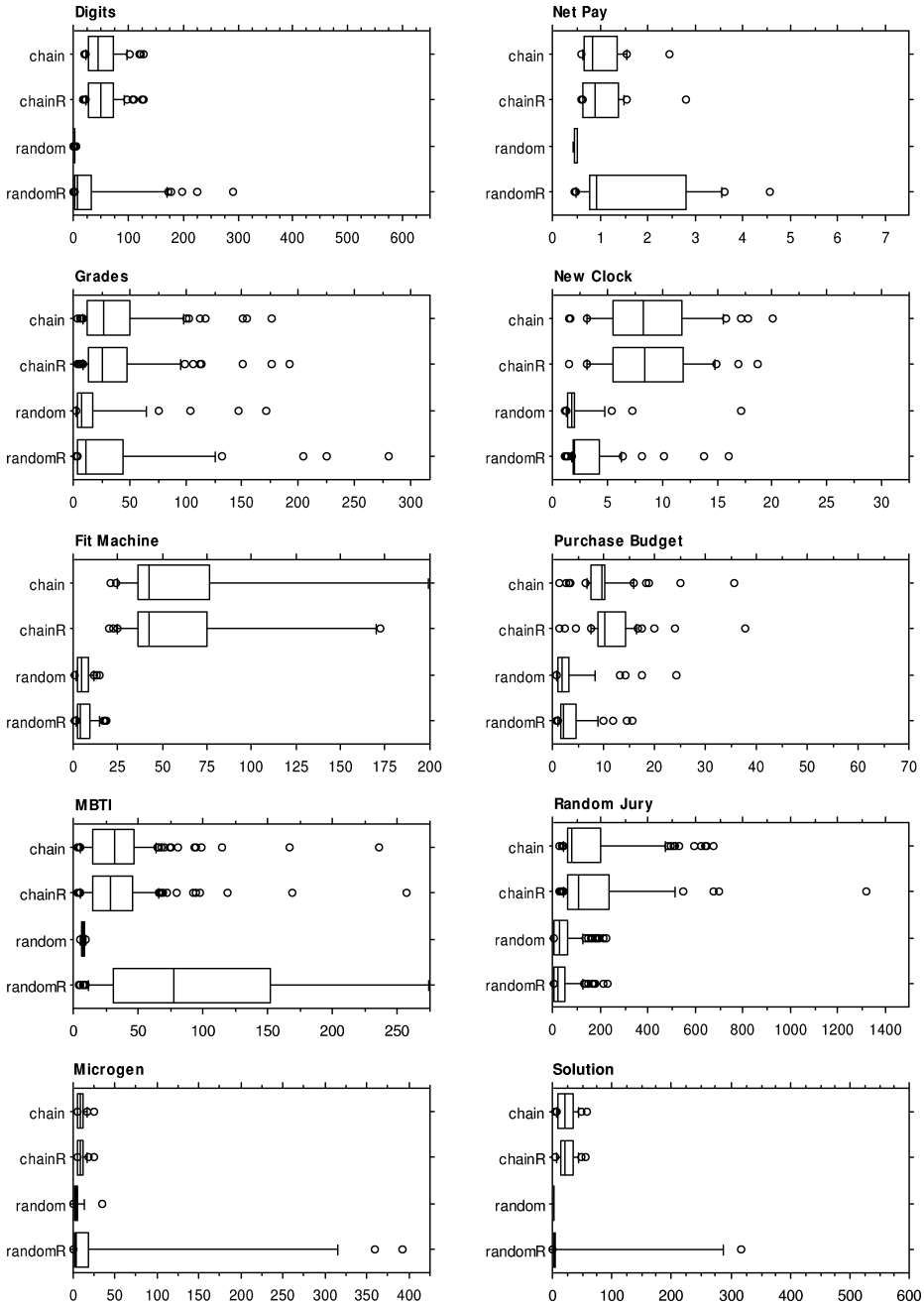

Fig. 12. Response times (seconds) for experiments at the du level. Technique abbreviations are: "chain" = Chaining without ranges, "chainR" = Chaining with ranges, "random" = Random without ranges, "randomR" = Random with ranges.

Table VIII. Median Response Times per Spreadsheet, Cell-Level Experiments

	Random without Ranges	Random with Ranges	Chaining without Ranges	Chaining with Ranges
Budget	2.4	3.0	13.1	14.4
Digits	2.1	7.8	44.4	42.4
FitMachine	3.4	3.3	53.3	53.4
Grades	6.0	16.5	20.7	25.3
MBTI	13.9	26.4	20.1	20.0
MicroGen	3.8	3.8	11.5	11.4
NetPay	1.3	1.3	1.1	1.1
NewClock	2.8	3.0	12.8	14.1
RandomJury	28.9	23.6	82.8	109.7
Solution	2.9	2.9	22.7	24.0

Table IX. Median Response Times per Spreadsheet, Du-Level Experiments

	Random without Ranges	Random with Ranges	Chaining without Ranges	Chaining with Ranges
Budget	2.0	2.2	9.7	10.1
Digits	1.8	8.8	46.6	50.6
FitMachine	4.5	3.9	41.7	42.7
Grades	6.6	11.3	27.1	25.6
MBTI	7.0	76.7	31.3	28.7
MicroGen	2.6	2.7	8.0	8.5
NetPay	0.5	0.9	0.8	0.8
NewClock	1.8	2.1	8.3	8.3
RandomJury	39.9	22.6	86.6	91.1
Solution	1.2	2.0	21.4	22.3

the boxplots of cell and du-association arrow-level data. The same effect occurs for `Random`, but less frequently.

4.6.3 *Research Question 3: Response Time on Failure.* The final research question we consider asks how long test generation will take to respond, given that the call to a test generation technique ends in failure, with no du-associations being exercised through that call. Recall, however, that `Random` cannot be meaningfully investigated in this context because its stopping time is hard-coded as a constant. Thus, we focus on `Chaining`.

Figure 13 presents boxplots showing all response data gathered on unsuccessful test generation attempts by `Chaining` with and without ranges, per spreadsheet, with results at all three levels represented. For ease of reference, Table X lists median response times.

As the table shows, median response times on failure varied widely across both spreadsheets and levels of application. The only clear trend that emerges from the data involves the fact that the use of range information only infrequently resulted in differences in failure response times. Median times differed by more than a few seconds at the spreadsheet-level on only three spreadsheets, and at the cell-level on only one. On `RandomJury`, `Chaining` required almost 100 times more seconds to respond without ranges than with ranges, however, in this case this turns out to be not particularly interesting, because through

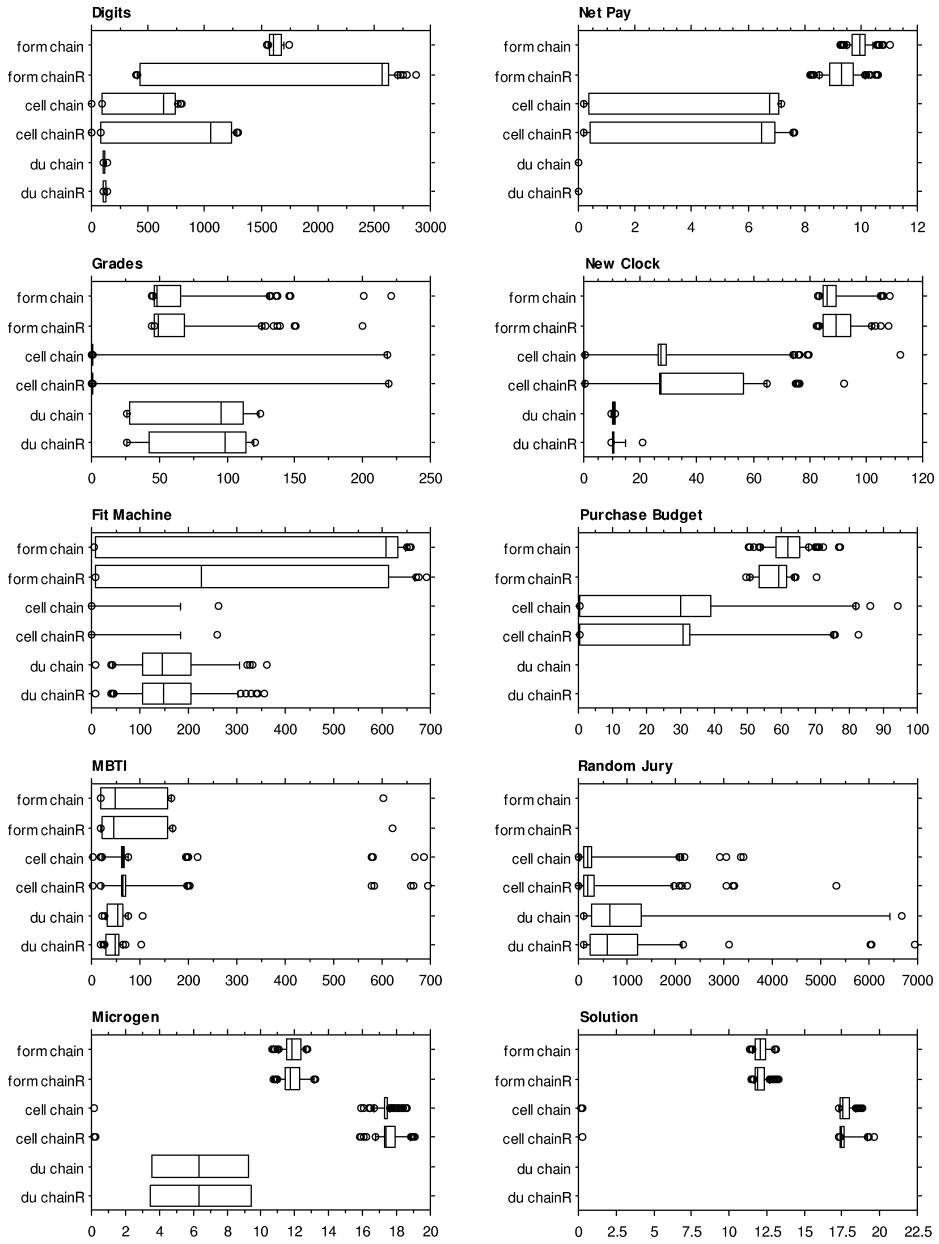

Fig. 13. Response times (seconds) for failing runs of `Chaining`, with and without ranges, at all levels of application. Technique abbreviations are: "chain" = Chaining without ranges, "chainR" = Chaining with ranges, "random" = Random without ranges, "randomR" = Random with ranges.

all of the form experiments on `RandomJury`, `Chaining` without ranges failed on only a single run. The similarity persists in most cases in the boxplots (i.e., the distributions of failure response times are similar with and without range information).

Table X.

	Spreadsheet		Cell		Du-Assoc. Arrow	
	Chaining without Ranges	Chaining with Ranges	Chaining without Ranges	Chaining with Ranges	Chaining without Ranges	Chaining with Ranges
Budget	61.6	59.3	30.1	30.3	—	—
Digits	1612.6	2559.1	704.2	1050.8	222.3	231.1
FitMachine	612.5	226.3	0.4	0.3	148.0	149.3
Grades	48.3	48.9	0.9	0.4	98.8	99.5
MBTI	34.4	46.2	81.2	80.7	53.8	48.5
MicroGen	11.8	11.8	17.2	17.3	6.3	6.3
NetPay	9.9	9.3	6.7	6.4	0.1	0.1
NewClock	86.4	89.2	27.0	27.1	10.2	10.1
RandomJury	1303.0	15.1	201.0	200.5	611.2	548.2
Solution	12.1	11.9	17.5	17.4	—	—

Median response times on failure for `chaining` at the spreadsheet, cell, and du-association arrow-levels. The notation "—" indicates that no failures occurred.

Of greater interest is the relationship between failure response times and the success response times listed in Table VII. In almost all cases, median response time for failure was higher than median response time for success (sometimes nearly 100 times higher). Exceptions involved `MBTI` without range information at the spreadsheet-level; `FitMachine`, `Grades` and `Solution` with and without range information at the cell-level; and `MicroGen` and `NetPay` with and without range information at the du-association arrow-level. However, the differences for the exceptional cases were small.

4.7 Discussion

Keeping in mind the limitations imposed by the threats to validity described in Section 4.5, the foregoing results have several implications. They also help us discover differences and trade-offs between techniques and application approaches. In several cases, these discoveries contradicted our initial expectations.

The primary implication of the results is that, at least from the point of view of effectiveness, automated test generation for spreadsheets with integer inputs is feasible. In the cases we considered, `Chaining` was highly effective (both with and without range information) at generating test inputs, achieving 100% coverage of feasible du-associations on half of the spreadsheets considered, greater than 97% coverage on all but one spreadsheet, and greater than 92% coverage on that one spreadsheet. A second aspect of feasibility, whether users can in fact use the system, must still be addressed, and this includes examining whether system response times are acceptable to those users.

A second implication involves choice of techniques. We had initially conjectured that with spreadsheets, random test generation might perform nearly as well as more complex heuristics, thus providing a more easily implemented approach to test generation. Our experiments suggest that, where effectiveness is concerned, this conjecture is false. In the cases we considered, `Random` techniques were much less effective at covering du-associations in spreadsheets than `Chaining` techniques, over half of the time achieving less than 80%

coverage. Further, Random techniques were much less consistent than Chaining techniques in terms of effectiveness: Whereas Chaining's effectiveness ranged only from 92% to 100% coverage, the effectiveness of Random techniques ranged from 25% to 100% coverage, a range nine times that of Chaining techniques.

Where response times are concerned, however, Random often performed considerably better than Chaining at lower levels of coverage, its efficiency benefits disappearing only as generation begins to focus on more difficult-to-cover du-associations. This result suggests that a hybrid approach combining both techniques might be more useful than an approach using either technique alone. Such a hybrid approach would apply a Random technique initially until a certain level of coverage has been achieved or until response times for Random exceed a certain threshold, and then switch over to using Chaining.

A third implication involves the use of range information. At the outset of this work we had postulated that provision of range information would benefit both test generation techniques. Where Random was concerned, this proved correct: Random with ranges was typically far more effective at achieving high levels of coverage than Random without ranges. In contrast, Chaining did not benefit, in terms of effectiveness, from the provision of range information, and in fact, Chaining without range information was marginally better than Chaining with range information at achieving higher coverage levels. On reflection, we suspect that the Chaining algorithm, restricted by ranges, is less able than its unrestricted counterpart to jump beyond local minima/maxima and find solutions. Range information also did not seem to improve technique response time for either Random or Chaining.

A fourth implication involves failure response times. The data collected here suggest that failure response times for Chaining vary considerably across spreadsheets, and sometimes can be quite large. This suggests that even though Chaining necessarily terminates in this application, a time limit on its run might need to be set, and this time limit may need to be calculated relative to certain characteristics (e.g., of complexity, or degree of coverage achieved thus far) of the spreadsheet being tested. However, in the cases we observed failure response times typically exceeded success response times, so we can expect that appropriate time limits would not greatly reduce the overall effectiveness of the technique.

Finally, we had expected that, called on to generate test inputs for more focused areas of a spreadsheet (individual cells or du-associations), test generation techniques could respond more quickly than when invoked with the entire spreadsheet as target. Comparing our results on these three levels, however, we find that although substantive differences could occur between techniques on specific spreadsheets, for the most part, these differences were not uniform across spreadsheets. The single exception to this involved the tendency of response times to occur more consistently near the median for Chaining at the cell- and du-levels than at the spreadsheet-level. This tendency might suggest that end users would find the cell-and du-levels preferable to the spreadsheet-level, and that for best system responsiveness, a user of the test generation facility using Chaining should be encouraged by the environment's user

interface to select an arrow or cell. In any case, the results suggest that all three levels of test generation may have utility.

4.8 Scalability

So far, our empirical work has focused on relatively small spreadsheets. The survey of spreadsheets described in Section 3.4 found that 3% of this collection of spreadsheets are of similar size (fewer than 50 cells) [Fisher and Rothermel 2005]. Therefore, it is important to consider how our methods might scale to larger spreadsheets.

Spreadsheets are often constructed through the replication of formulas across many cells. Thus, we have developed methods in WYSIWYT to lower the cost of testing by grouping together cells with the same formulas into regions that can then share information from testing decisions [Burnett et al. 2002]. With the ability to share testing decisions between cells within regions, our test generation methodology then only needs to consider a representative of these regions in order to generate test inputs to achieve coverage of the spreadsheet.

To determine how much actual generation might be needed for larger spreadsheets, we examined the corpus of Excel spreadsheets collected in our survey. From the collection, we considered 370 spreadsheets that included conditionals (`if`, `choose`, `sumif`, and `countif`), since without conditionals there is no need to generate test inputs to achieve coverage under the WYSIWYT methodology. Within these spreadsheets we counted the number of distinct formulas (two formulas are defined as distinct if they could not have resulted from a copy/paste or fill operation without additional user editing). Among the 370 spreadsheets, 227 had fewer than 30 distinct formulas, the number of formula cells in MBTI. Therefore, it seems possible that our test generation techniques could scale to many "real world" spreadsheets.

5. CONCLUSIONS AND FUTURE WORK

We have presented an automated test generation methodology for spreadsheet languages. Our methodology is driven by the end user's request, allowing the user to generate test inputs for a specific du-association or cell, or at the whole-spreadsheet-level. Our methodology is tightly integrated with the highly interactive spreadsheet programming environment, presenting test data visually. We have utilized two test generation techniques within our methodology, one based on random generation and the second based on Ferguson and Korel's dynamic, goal-oriented approach. The details of these techniques, however, do not need to be known by the end users of our system in order for them to use them. We have prototyped our methodology, and our empirical studies suggest that it can effectively and efficiently generate test inputs.

Our results support several suggestions about the use of, and future research on, automated test generation for spreadsheets:

—If one can implement only one technique, and effectiveness is the primary consideration, one should implement the goal-oriented technique rather than the random technique.

—If one can implement only random test generation, one should make provisions for providing range information.
—A hybrid approach that begins with random generation to decrease response time during earlier generation attempts, and then utilizes a goal-oriented technique to reach higher levels of coverage might be better than either approach alone.
—All three levels of test generation proposed can conceivably be useful, and there are no strong reasons to prefer any at present; hence, a deciding factor among these may be user preference.

The approach to test case generation described here is just one component of a larger effort to bring more formal software engineering practices to end-users in the context of spreadsheets. Our goal is to seamlessly integrate various dependability tools to form a methodology for the development of dependable spreadsheets. Our other work on other aspects of this methodology relates to the work described in this article in various ways, as follows.

We have developed a methodology for reusing tests after spreadsheet modifications [Fisher et al. 2002]. As the user tests and edits the spreadsheets, tests that are impacted by edits are saved in a pool of potentially rerunnable tests. The Help-Me-Test mechanism then provides a test from this pool if possible, and only when not possible does it generate a new test using the methods described here.

In addition, we have developed an assertion mechanism in Forms/3 [Burnett et al. 2003]. Users are able to enter ranges of values for cells. These ranges, when on input cells, are used to limit the search process used by our test generation methodologies. When the ranges are placed on cells with formulas, our test generation algorithms can be used to find violating test cases (test cases that result in values outside of the user-supplied ranges) which indicate the presence of a fault in either the spreadsheet formulas or the supplied range. We also use the values generated by our test generation process to encourage users to enter assertions via a methodology we call "Surprise-Explain-Reward" [Wilson et al. 2003].

We have researched many ways in which to reduce the number of necessary du-associations to achieve our coverage criterion. A significant reduction may have an effect on the efficiency of automated test case creation techniques. To this end, we considered taking advantage of research that minimizes the all du-associations coverage criterion, such as via minimum spanning sets [Marre and Bertolino 1996a, 1996b]. Minimum spanning sets take into consideration implied coverage. For example, if covering du-association (d,u) implies covering du-association (d',u'), it may be sufficient to explicitly cover only du-association (d,u) to achieve all du-associations coverage. We have pursued this line of work in related research adapting WYSIWYT to a language based on the screen transition diagram paradigm [Brown et al. 2003], and a similar approach could be applied within WYSIWYT.

Given these results, our next step in this research is the design and performance of additional experiments, including experiments with a wider range of spreadsheets involving representatives of commercial spreadsheet applications

and using additional data types, and experiments involving human users of our methodology. The results presented in this article motivate the choice, for that experimentation, of either the `Chaining` technique for test generation, or of a hybrid technique using `Random` initially, followed by `Chaining`, in either case requiring no provision of range information. Such studies will help us assess whether our methodology can be used effectively by end users on production spreadsheets.

Alternatively, although `Chaining` managed to achieve a great deal of coverage with reasonable performance, any of several more recent approaches to test case generation in imperative programs, including the use of genetic algorithms [Michael et al. 2001], constraint logic programming [Gotlieb et al. 2000], and consistency techniques [Sy and Deville 2003], could be adapted to our test case generation framework.

ACKNOWLEDGMENTS

We thank the Visual Programming Research Group for their work on Forms/3 and their feedback on our methodologies.

REFERENCES

AVRITZER, A. AND WEYUKER, E. 1995. Automated generation of load test suites and the assessment of the resulting software. *IEEE Trans. Softw. Eng. 21,* 9 (Sept.), 705–716.

BARESEL, A., BINKLEY, D., HARMAN, M., AND KOREL, B. 2004. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis* (Boston, Mass). ACM Press, New York, 108–118.

BELKIN, N. 2000. Helping people find what they don't know. *Commun. ACM 41,* 8 (Aug.), 58–61.

BIRD, D. AND MUNOZ, C. 1983. Automatic generation of random self-checking test cases. *IBM Syst. J. 22,* 3, 229–245.

BOYAPATI, C., KHURSHID, S., AND MARINOV, D. 2002. Korat: Automated testing based on Java predicates. In *Proceedings of the ACM International Symposium on Software Testing and Analysis* (Rome). ACM Press, New York, 123–133.

BROWN, D., BURNETT, M., ROTHERMEL, G., FUJITA, H., AND NEGORO, F. 2003. Generalizing WYSIWYT visual testing to screen transition languages. In *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments* (Auckland, New Zealand). IEEE Computer Society Press, Los Alamitos, Calif.

BROWN, P. AND GOULD, J. 1987. Experimental study of people creating spreadsheets. *ACM Trans. Office Inf. Syst. 5,* 3 (July), 258–272.

BURNETT, M., ATWOOD, J., DJANG, R., GOTTFRIED, H., REICHWEIN, J., AND YANG, S. 2001. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Functional Program. 11,* 2, 155–206.

BURNETT, M., COOK, C., PENDSE, O., ROTHERMEL, G., SUMMET, J., AND WALLACE, C. 2003. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the International Conference on Software Engineering* (Portland, Oreg.). IEEE Computer Society Press, Los Alamitos, Calif. 93–103.

BURNETT, M., COOK, C., AND ROTHERMEL, G. 2004. End-user software engineering. *Commun. ACM 47,* 9 (Sept.), 53–58.

BURNETT, M., HOSSLI, R., PULLIAM, T., VANVOORST, B., AND YANG, X. 1994. Toward visual programming languages for steering in scientific visualization: A taxonomy. *IEEE Comput. Sci. Eng. 1,* 4, 44–62.

BURNETT, M., SHERETOV, A., REN, B., AND ROTHERMEL, G. 2002. Testing homogeneous spreadsheet grids with the "What You See Is What You Test" methodology. *IEEE Trans. Softw. Eng. 28,* 6 (June), 576–594.

BURNETT, M., SHERETOV, A., AND ROTHERMEL, G. 1999. Scaling up a "What You See Is What You Test" methodology to spreadsheet grids. In *Proceedings of the 1999 IEEE Symposium on Visual Languages* (Tokyo). IEEE Computer Society Press, Los Alamitos, Calif. 30–37.

CHANG, J. AND RICHARDSON, D. 1999. Structural specification-based testing: Automated support and experimental evaluation. In *Proceedings of the ACM Symposium on Foundations of Software Engineering* (Toulouse, France). ACM Press, New York, 285–302.

CHI, E., BARRY, P., RIEDL, J., AND KONSTAN, J. 1997. A spreadsheet approach to information visualization. In *Proceedings of the IEEE Symposium on Information Visualization* (Phoenix, Ariz). IEEE Computer Society Press, Los Alamitos, Calif.

CLARKE, L. 1976. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng. 2,* 3 (Sept.), 215–222.

CORRITORE, C., KRACHER, B., AND WIEDENBECK, S. 2001. Trust in the online environment. In *Proceedings of the HCI International* (New Orleans, La). Lawrence Erlbaum, Mahwah, N.J., 1548–1552.

CULLEN, D. 2003. Excel snafu costs firm $24m. *The Register*.

DEMILLO, R. AND OFFUTT, A. 1991. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng. 17,* 9 (Sept.), 900–910.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. 1992. Rigorous data flow testing through output influences. In *Proceedings of the 2nd Irvine Software Symposium* (Irvine, Calif). University of California, Irvine.

ERNST, M., COCKRELL, J., GRISWOLD, W., AND NOTKIN, D. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng. 27,* 2 (Feb.), 1–25.

FERGUSON, R. AND KOREL, B. 1996. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol. 5,* 1 (Jan.), 63–86.

FISHER, M., II, CAO, M., ROTHERMEL, G., COOK, C., AND BURNETT, M. 2002. Automated test case generation for spreadsheets. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Fla.). ACM Press, New York, 241–251.

FISHER, M., II, JIN, D., ROTHERMEL, G., AND BURNETT, M. 2002. Test reuse in the spreadsheet paradigm. In *Proceedings of the International Symposium on Software Reliability Engineering* (Annapolis, Md.). IEEE Computer Society, Los Alamitos, Calif.

FISHER, M., II AND ROTHERMEL, G. 2005. The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Proceedings of the Workshop on End-User Software Engineering* (St. Louis, Miss.). ACM Press, New York.

FRANKL, P. AND WEYUKER, E. 1988. An applicable family of data flow criteria. *IEEE Trans. Softw. Eng. 14,* 10 (Oct.), 1483–1498.

GOTLIEB, A., BOTELLA, B., AND REUHER, M. 1998. Automatic test data generation using constraint solving techniques. In *Proceedings of the ACM International Symposium on Software Testing and Analysis* (Clearwater Beach, Fla.). ACM Press, New York, 53–62.

GOTLIEB, A., BOTELLA, B., AND RUEHER, M. 2000. A CLP framework for computing structural test data. In *Proceedings of the 1st International Conference on Computational Logic* (London). Springer Verlag, New York, 399–412.

GUPTA, N., MATHUR, A., AND SOFFA, M. 1998. Automated test data generation using an iterative relaxation method. In *Proceedings of the International Symposium on Foundations of Software Engineering* (Lake Buena Vista, Fla.). ACM Press, New York, 231–244.

HARTMAN, A. AND NAGIN, K. 2003. Model driven testing—Agedis architecture interfaces and tools. In *Proceedings of the 1st European Conference on Model Driven Software Engineering* (Nuremberg). imbus AG, Möhrendorf, Germany, 1–11.

KOREL, B. 1990a. A dynamic approach of automated test data generation. In *Proceedings of the International Conference on Software Maintenance* (San Diego, Calif.). IEEE Computer Society Press, Los Alamitos, Calif., 311–317.

KOREL, B. 1990b. Automated software test data generation. *IEEE Trans. Softw. Eng. 16,* 8 (Aug.), 870–897.

KRISHNA, V., COOK, C., KELLER, D., CANTRELL, J., WALLACE, C., BURNETT, M., AND ROTHERMEL, G. 2001. Incorporating incremental validation and impact analysis into spreadsheet maintenance: An empirical study. In *Proceedings of the International Conference on Software Maintenance* (Florence). IEEE Computer Science Press, Los Alamitos, Calif.

LASKI, J. AND KOREL, B. 1993. A data flow oriented program testing strategy. *IEEE Trans. Softw. Eng. 9,* 3 (May), 347–354.

LEOPOLD, J. AND AMBLER, A. 1997. Keyboardless visual programming using voice, handwriting, and gesture. In *Proceedings of the 1997 IEEE Symposium of Visual Languages* (Capri, Italy). IEEE Computer Society Press, Los Alamitos, Calif., 28–35.

MARINOV, D. AND KHURSHID, S. 2001. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the International Conference on Automated Software Engineering* (San Diego, Calif.). IEEE Computer Society Press, Los Alamitos, Calif.

MARRE, M. AND BERTOLINO, A. 1996a. Reducing and estimating the cost of test coverage criteria. In *Proceedings of the International Conference on Software Engineering* (Berlin). IEEE Computer Society Press, Los Alamitos, Calif.

MARRE, M. AND BERTOLINO, A. 1996b. Unconstrained duas and their use in achieving all-uses coverage. In *Proceedings of the ACM International Symposium on Software Testing and Analysis* (San Diego, Calif.). ACM Press, New York.

MICHAEL, C., MCGRAW, G., AND SHATZ, M. 2001. Generating software test data by evolution. *IEEE Trans. Softw. Eng. 27,* 12 (Dec.), 1085–1110.

MYERS, B. 1991. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of the ACM CHI '91* (New Orleans, La). ACM Press, New York, 243–249.

OFFUTT, A. 1991. An integrated automatic test data generation system. *J. Syst. Integration 1,* 3 (Nov.), 391–409.

OFFUTT, J. AND ABDURAZIK, A. 1999. Generating tests from uml specifications. In *Proceedings of the International Conference on the Unified Modeling Language* (Fort Collins, Colo.). Springer Verlag, New York.

PANDE, H., LANDI, W., AND RYDER, B. 1994. Interprocedural def-use associations in C programs. *IEEE Trans. Softw. Eng. 20,* 5 (May), 385–403.

PANKO, R. 1995. Finding spreadsheet errors: Most spreadsheet errors have design flaws that may lead to long-term miscalculation. *Information Week*, 100.

PANKO, R. 1998. What we know about spreadsheet errors. *J. End User Comput. 10,* 2 (Spring), 15–21.

RAMAMOORTHY, C., HO, S., AND CHEN, W. 1976. On the automated generation of program test data. *IEEE Trans. Softw. Eng. 2,* 4 (Dec.), 293–300.

RAPPS, S. AND WEYUKER, E. 1985. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. 11,* 4 (Apr.), 367–375.

RAZ, O., KOOPMAN, P., AND SHAW, M. 2002. Semantic anomaly detection on online data sources. In *Proceedings of the International Conference on Software Engineering* (Orlando, Fla). ACM Press, New York, 302–312.

ROTHERMEL, G., BURNETT, M., LI, L., DUPUIS, C., AND SHERETOV, A. 2001. A methodology for testing spreadsheets. *ACM Trans. Softw. Eng. Methodol. 10,* 1 (Jan.), 110–147.

ROTHERMEL, G., LI, L., AND BURNETT, M. 1997. Testing strategies for form-based visual programs. In *Proceedings of the 8th International Symposium on Software Reliability Engineering* (Albuquerque, N. Mex.). IEEE Computer Society Press, Los Alamitos, Calif., 96–107.

ROTHERMEL, G., LI, L., DUPUIS, C., AND BURNETT, M. 1998. What You See Is What You Test: A methodology for testing form-based visual programs. In *Proceedings of the 20th International Conference on Software Engineering* (Kyoto). IEEE Computer Society Press, Los Alamitos, Calif., 198–207.

ROTHERMEL, K., COOK, C., BURNETT, M., SCHONFELD, J., GREEN, T., AND ROTHERMEL, G. 2000. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In *Proceedings of the 22nd International Conference on Software Engineering* (Limerick, Ireland). ACM Press, New York.

SCOTT, A. 2003. Shurgard stock dives after auditor quits over company's accounting. *The Seattle Times*.

SMEDLEY, T., COX, P., AND BYRNE, S. 1996. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *Proceedings of the Conference on Advanced Visual Interfaces '96* (Gubbio, Italy). ACM Press, New York.

SMITH, R. 2004. University of Toledo loses $2.4m in projected revenue. *The Toledo Blade*.

SY, N. AND DEVILLE, Y. 2003. Consistency techniques for interprocedural test data generation. In *Proceedings of the ESEC/FSE'03* (Helsinki). ACM Press, New York, 108–117.

VIEHSTAEDT, G. AND AMBLER, A. 1992. Visual representation and manipulation of matrices. *J. Visual Lang. Comput. 3,* 3 (Sept.), 273–298.

VISSER, W., PASAREANU, C., AND KHURSHID, S. 2004. Test input generation with Java Pathfinder. In *Proceedings of the ACM International Symposium on Software Testing and Analysis* (Boston, Mass). ACM Press, New York, 97–107.

WEYUKER, E. 1993. More experience with dataflow testing. *IEEE Trans. Softw. Eng. 19,* 9 (Sept.), 912–919.

WILCOX, E., ATWOOD, J., BURNETT, M., CADIZ, J., AND COOK, C. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proceedings of the ACM CHI'97* (Atlanta, Ga). ACM Press, New York, 22–27.

WILSON, A., BURNETT, M., BECKWITH, L., GRANATIR, O., CASBURN, L., COOK, C., DURHAM, M., AND ROTHERMEL, G. 2003. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (Ft. Lauderdale, Fla). ACM Press, New York, 305–312.

Received October 2002; revised March 2005; accepted October 2005