Integrating Automated Test Case Generation into the WYSIWYT Spreadsheet Testing Methodology*

M. Fisher II, M. Cao, G. Rothermel, D. Brown, C. R. Cook, M. Burnett
Department of Computer Science
Oregon State University
grother@cs.orst.edu

September 29, 2002

Abstract

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 assists spreadsheet users in testing their spreadsheet formulas. Our empirical studies have shown that this methodology can help end users test spreadsheets more adequately and efficiently; however, the process of generating test cases can still represent 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 utilized two techniques for generating test cases, one involving random selection and another 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.

1 Introduction

Spreadsheet languages are widely used by a variety of end users to perform important tasks, such as tax calculations, budget management, and quality assessments of pharmaceutical products. The spreadsheets these end users create steer important decisions that may affect the welfare or safety of individuals, or even of large segments of society. The spreadsheet language paradigm is also a subject of ongoing research; for example, there is research into using spreadsheet languages for matrix manipulation problems [32], for scientific visualization [8], for providing steerable simulation environments for scientists [5], and for specifying full-featured GUIs [22, 31].

It is important that spreadsheets function correctly, but research shows that they often contain faults. A survey of the literature [25] provides details: in four field audits of operational spreadsheets, faults were found in 20.6% of the spreadsheets audited; in eleven experiments in which participants created spreadsheets, faults were found in an average of 60.8% of those spreadsheets; in four experiments in which participants inspected spreadsheets for faults, an average of 55.8% of those faults were missed. Compounding these problems is the unwarranted confidence spreadsheet users have in the correctness of their spreadsheets [3, 34].

*This paper is a revised and expanded version of a paper presented at the International Conference on Software Engineering, May, 2002 [14].
In spite of such evidence, until recently, little work had been done to help end users assess the correctness of their spreadsheets. Thus, we have been developing a testing methodology for spreadsheets termed the “What You See Is What You Test” (WYSIWYT) methodology [27, 28, 29]. 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 users test their spreadsheets more adequately and more efficiently, and also reduce end-user overconfidence [20, 30]. In subsequent work, we extended this approach to apply to large grids of cells [6, 7], and to incorporate re-use of test cases [15].

As presented to date, the WYSIWYT methodology has relied solely on the intuitions of spreadsheet users to identify test cases for their spreadsheets. In general, the process of manually identifying 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 case generation (e.g., [9, 11, 13, 17, 18, 19, 26]), however, has been directed at imperative languages, and we can find no research specifically addressing automated test case generation for spreadsheet languages.

To address this problem, we have been investigating how to automate the generation of test cases for spreadsheets in ways that support incremental testing and provide immediate visual feedback. We have utilized two techniques for test case generation: one using random selection and one using a goal-oriented approach [13]. We describe these techniques and their integration into our WYSIWYT methodology, and report results of experiments examining their effectiveness and efficiency.

In the next section of this paper we present background material on the spreadsheet programming paradigm, the WYSIWYT testing methodology, and test case generation techniques for imperative programs. Section 3 discusses the requirements for an automatic test case generation technique for spreadsheet languages, and then presents our techniques based on that discussion. Section 4 presents the design and results of several experiments we have performed using our testing methodology, and discusses 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. [4, 8, 21, 31]) based on this paradigm.

In this article, we present examples of spreadsheets in the research language Forms/3 [4]. Figure 1 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 1: Forms/3 spreadsheet for calculating grades.

Figure 2: Forms/3 spreadsheet Budget.

Figure 2 shows a second example of a Forms/3 spreadsheet, Budget, which calculates how many pens and paper clips an office will have after an order and whether that order is within a given budget amount.

As the figures show, Forms/3 spreadsheets, like traditional spreadsheets, consist of cells, but these cells are not restricted to grids.
2.2 The WYSIWYT Methodology

In our “What You See Is What You Test” (WYSIWYT), methodology for testing spreadsheets [27, 29, 30], 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 validates the results displayed in those cells. Behind the scenes these validations are used to measure the quality of testing in terms of a dataflow adequacy criterion, which tracks coverage of interactions between cells caused by cell references.¹

The following example illustrates the process. Suppose the user constructs the Budget spreadsheet by entering cells and formulas, reaching the state shown in Figure 2. Note that at this point, all cells other than input cells have red borders (light gray in this paper), 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; thus, their borders are thin and black to indicate to the user that they are not testable.)

Suppose the user looks at the values displayed on the screen and decides that cell BudgetOK? contains the correct value, given the current input values. To communicate this fact, the user checks off the value by clicking on the decision box in the upper right corner of that cell. One result of this “validation” action, shown in Figure 3, is the appearance of a checkmark in the decision box, indicating that the cell’s output has been validated under current inputs. (Two other decision box states, empty and question mark, are possible: each indicates that the cell’s output has not been validated under the current inputs. In addition, the question mark indicates that validating the cell would increase testedness.)

¹Other applicable criteria are discussed in [28].
A second result of the user's "validation" action is that the colors of the validated cell's borders become more blue, indicating that interactions caused by references in that cell's formula have been exercised in producing validated outputs. In the example, in the formula for BudgetOK?, references in the else clause have now been exercised, but references in the then clause have not; thus, that cell's border is partially blue (slightly darker gray in this paper). Testing results also flow upstream in the dataflow to other cells whose formulas have been used in producing a validated value. In our example, all interactions ending in the cell ClipTotal have been exercised; hence, that cell's border is now fully blue (very dark gray in this paper).

If users choose, they can also view interactions caused by cell references by displaying dataflow arrows between cells or 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 granularity, following the same color scheme as for the cell 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 [12]; for brevity we call our adaptation of this criterion the du-adequacy criterion. We precisely define this criterion in [27]; here, we summarize that presentation.

The du adequacy criterion is defined through an abstract model of spreadsheets called a cell relation graph (CRG). Figure 4 shows the CRG for 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 non-predicate node that refers to C) or a predicate use (an out-edge from a predicate node that refers to C). A definition-use association (du-association) links a definition of C with a use of C which that definition can reach. A du-association is exercised by a test when inputs have been found that cause the expressions associated with its definition and its use to be executed, and where this execution produces a value in some cell that is pronounced "correct" by a user validation. Under the du-adequacy criterion, testing is adequate when each du-association in a spreadsheet has been exercised by at least one test.
In this model, a test case for a spreadsheet is a tuple \((I, C)\), where \(I\) is a vector of input values corresponding to input cells in the spreadsheet, and \(C\) is a cell whose value the user has validated under that input configuration. A test (the user’s act of applying a test case) is an explicit decision by the user that \(C\)’s value is correct, given the current configuration \(I\) of input cell values.

It is not always possible to exercise all du-associations in a spreadsheet; those that cannot be exercised by any inputs are called infeasible du-associations. In general, the problem of identifying such du-associations is undecidable [16, 33].

2.3 ATCG techniques for imperative languages

There has been considerable research on techniques for automatic test case generation (ATCG) for imperative languages. Ferguson and Korel [13] classify ATCG techniques according to their mechanism of generation into three categories: random, path-oriented and goal-oriented. We summarize their classification here.

Random test case generation techniques [2] generate test cases by randomly selecting input values.

Path-oriented test case generation techniques first select a program path that will meet a testing re-
quirement, 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 [9, 11, 17, 23] 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. A subclass of path-oriented techniques, execution-oriented techniques [19] 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 case generation techniques [13, 18], 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 the goal. Like execution-oriented methods, these 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 [13].

3 Automated Test Case Generation and WYSIWYT

In any testing methodology that does not automatically generate test cases, 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 case 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. Section 3.2 then shows how to extend this approach to incorporate Ferguson and Korel’s goal-oriented input generation technique, and Section 3.3 discusses a refinement involving provision of range information with spreadsheet cells.

3.1 Basic Methodology

Suppose a user desires help 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 the entire spreadsheet), then push the “Help Me Test” button in the Forms/3 toolbar. At this point the underlying test case generation system responds, attempting to generate a test case.

Figure 5 provides an overview of the process the test case generation system follows. The system’s first task (line 1) is to call CalculateUsefulDUs to determine the set Use fulDUs 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 indicated specific cells or du-association
algorithm GenerateTestCase(Cells, Arrows)
inputs   Cells : Cells indicated by user
          Arrows : Arrows indicated by user
1. UsefulDUs = CalculateUsefulDUs(Cells, Arrows)
2. InputCells = CalculateInputCells(UsefulDUs)
3. InputValues = SaveInputCells(InputCells)
4. if GenerateTestCase(UsefulDUs, InputCells) then
5.   UpdateDisplay()
6. else
7.   RestoreConfig(InputCells, InputValues)
8. end if

Figure 5: Overall algorithm for generating a test case.

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 node 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 case 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 [27]), it is available in data structures kept by the engine, and CalculateInputCells simply retrieves it from there.

Given UsefulDUs and InputCells, the test case generation system can attempt to generate a test case. The system first saves the existing configuration of input cell values (line 3) for restoration if generation fails, and then, through GenerateTestCase, invokes a test case generation technique (line 4). As Section 2.3 indicated, there are many test case generation techniques that could be utilized; the simplest of these is to randomly generate input values. To provide this technique, we have GenerateTestCase 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 subsequent evaluation causes any du-associations in UsefulDUs to be executed. If no such du-association is executed, Random repeats this process with a new set of random values, iterating until a set of values that executes a du-association of interest has been found, or until a built-in time limit is reached. As the system applies these new input values, the values appear in the spreadsheet itself and also in the “Help Me Test” window, along with messages detailing the activities of the system. (Displaying the values being tried carries a performance penalty, but an advantage is that it communicates to the user approximately what the system
is doing, an understanding of which is often a significant factor in users' effectiveness and continuing use of a system [1, 10].

If Random exercises a du-association in *UsefulDUs*, the system has generated a potential set of test inputs. However, this is not yet a test – recall that a test consists of the user validating an output value. Hence, the system now needs to communicate not only the generated test inputs to the user, but also which cell(s) that use this set of inputs can be validated. Even without automatic test case generation, the WYSIWYT system maintains information about the cells whose validation could increase testedness, and uses this to display advice to the user (in the form of question marks in decision boxes as detailed in Section 2.2). However, this information pertains to, and is displayed on, all cells in the spreadsheet. To direct the user to cells whose validation would increase the coverage of the elements the user selected, the test generation system determines the set of relevant validatable output cells resulting from the new test inputs, and presents a list of these, along with the input cells it ultimately changed to generate them, to the user. (The relevant validatable output cells are also displayed with question marks in the spreadsheet itself.) Relevant validatable output cells include the selected cells themselves, as well as downstream cells whose validation would cover the selected cells. For example, if the user requested help testing cell BudgetOK? in Figure 3, the system would manipulate the values in cells PenUG, PenUP, etc., and would present only BudgetOK? as the relevant validatable cell.

If the system succeeds in finding a useful set of test inputs and displaying validatable output cells, the user at this point can validate an output value. Alternatively, the user can ignore the generated values (e.g. if they dislike the input set generated) and choose to try again with the same or different cells or du-association arrows selected, in which case the test case generation technique tries again using new seeded values.

If, on the other hand, the system reaches its built-in time limit without Random 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 a test case. In this case, too, the user can try again and the system will begin with new seeded values.

### 3.2 Goal-oriented test case 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 case generation for imperative programs.

On the other hand, spreadsheets and the WYSIWYT approach lend themselves naturally to goal-oriented test case generation, which requires dynamic execution traces (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 case generation techniques. Ultimately, we adapted Ferguson and Korel's “Chaining Approach” [13] for our purpose; we call our adaptation Chaining.

---

2Personal communication, Jeff Offutt.
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 Executed(d,u) then
12.    return success
13. end if
14. end while
15. end for
16. return failure

Figure 6: Chaining algorithm.

3.2.1 Overall algorithm

The Chaining algorithm is presented in Figure 6. Like Random, Chaining is invoked to find a set of inputs that exercise one or more du-associations in UsefulDUs. In terms of Figure 5, we simply have GenerateTestCase (line 5) 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 valditable 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 case.

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 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.3 For example, to exercise du-association (24,(33,T)) in Budget (see the CRG in Figure 4 and its associated spreadsheet in Figure 3), 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 a constraint path in the cell formula graph for the cell containing d (or u), consisting of the entry node e for that formula graph, the appropriate edge out of each predicate node lying on the direct path from e to d (or u), and d (or u). For example, the constraint path for definition 24 and use (33, T) in the CRG for Budget

---

3An additional requirement present for imperative programs — that the definition “reach” 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 [27].
are $(22, (23, T), 24)$ and $(32, (33, T))$, respectively. The constraint path for du-association $(d, u)$ consists of the concatenation of the constraint paths for $d$ and $u$. Thus, for example, the constraint path for du-association $(24, (33, T))$ in **Budget** is $(22, (23, T), 24, 32, (33, T))$.

When considering du-association $(d, u)$, **Chaining** first constructs the constraint path for $(d, u)$. Given our methodology, it is necessary that under the current inputs, $(d, u)$ 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 the constraint path are being evaluated in a manner that causes nodes in the constraint path to not be reached. **Chaining**’s task is to alter this situation, by finding inputs that cause all nodes on the constraint path to be reached.

To do this, **Chaining** compares the constraint path for $(d, u)$ to the path built by concatenating the execution traces for the cells containing $d$ and $u$. These execution traces consist of the lists of CRG nodes and edges executed in the cells during the cells’ most recent evaluations, and they can be retrieved from the spreadsheet engine, which previously collected them for use by the WYSIWYT subsystem. **Chaining** then identifies the break point in the constraint path: the first predicate in the execution trace that proceeds down a different branch than the same predicate in the constraint path (or, less formally, the earliest “incorrectly taken branch” in the execution traces).

In Figure 6, calculating the constraint path and finding the break point occurs in the **FindBreakPoint** function invoked in line 4. In the example we have been considering, the concatenated execution trace, assuming the spreadsheet’s input cells have the values shown in Figure 3, is $(22, (23, F), 25, 26, 32, (33, F), (35, F), 37, 38)$, and the break point is $(23, F)$.

Given a break point $b$, **Chaining**’s next task is to find inputs that cause the predicate in $b$ 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 we attempt to “chain back” from $b$, a procedure described shortly. If both the search and the “chaining” process fail, we have failed on our attempt to cover this du-association, and we move on to the next du-association in **UsefulDUs** (line 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 in $b$ 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 which 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 \(\text{TotalCost}\) 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 \((p, T)\).

When a problem break point is encountered, \(\text{Chaining}\) cannot make progress on the current break point. It is possible, however, that by exercising some other du-association that influences the outcome of the predicate in the break point, \(\text{Chaining}\) will be able to make progress. Thus, faced with a problem break point \(b\), \(\text{Chaining}\) calls \(\text{ChainDU}\)s (line 7), which collects a set \(\text{ChainDU}\)s of other du-associations \((d', u')\) in the spreadsheet that have two properties: (1) \(u'\) is the predicate use associated with the alternate branch of \(b\), 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. \(\text{Chaining}\) iterates through du-associations in \(\text{ChainDU}\)s, applying (recursively) the same process described for use on \((du)\) to each. As discussed in [13], 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 \((23,F)\) is designated a problem break point, i.e. the search algorithm was not able to force node 23 to take the "true" branch. In this case, \(\text{Chaining}\) would proceed to "chain back" from node 23. It would then set \(\text{ChainDU}\)s to \((29,(23,T))\), and try to exercise this du-association.

### 3.2.2 The search procedure

The search procedure used by \(\text{Chaining}\) to find inputs that cause predicates to take alternative branches involves two steps. First, a branch function is created, based on the predicate, to guide the search, and 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 7.

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 [13], as shown in Table 1. 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) 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 2. The purpose of these functions is to allow other branch functions to be combined in a meaningful way.
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

Figure 7: Algorithm for minimizing the branch function for a break point

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Branch Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l &lt; r )</td>
<td>( r - l )</td>
</tr>
<tr>
<td>( l &gt; r )</td>
<td>( l - r )</td>
</tr>
<tr>
<td>( l \leq r )</td>
<td>if ( r - l \geq 0 ) then ( r - l + 1 ) else ( r - l )</td>
</tr>
<tr>
<td>( l \geq r )</td>
<td>if ( l - r \geq 0 ) then ( l - r + 1 ) else ( l - r )</td>
</tr>
<tr>
<td>( l = r )</td>
<td>if ( l = r ) then ( 1 ) else (</td>
</tr>
<tr>
<td>( l \neq r )</td>
<td>(</td>
</tr>
</tbody>
</table>

Table 1: Branch functions for true branches of relational operators.

After calculating the branch function for a break point, the search procedure seeks a set of inputs that satisfy 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 [13], 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 act on the results of successful exploratory moves, incrementing or decrementing values of a candidate input (by potentially 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.
<table>
<thead>
<tr>
<th>Logical Operator</th>
<th>Branch Function</th>
</tr>
</thead>
</table>
| $l$ 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: $f(e, true)$

Table 2: Branch functions for logical operators.

In the example that we have been using, the branch function for the break point (23, F) is “if UnitsError – 1 = 0 then 1 else –[UnitsError – 1].” The input cells that could affect this branch function are ClipU0 and PenU0. Assume that the search procedure first considers the cell PenU0, and that the input cells have initial values as shown in Figure 3. In this case, the branch function evaluates to -1 (line 7). First the search procedure performs exploratory moves to try to determine which direction the input should change (lines 9-19). PenU0 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 ClipU0, with similar results. At this point the search procedure returns and node 23 is declared a problem node.

Since node 23 is a problem node, Chaining next considers the du-association, (29, (23,T)), which has (28,F) as its break point. The branch function for this break point is “if PenU0 – 0 \leq 0 and ClipU0 – 0 \leq 0 then 0 – PenU0 +0 – ClipU0 else 1.” With the initial inputs from Figure 3, this branch function evaluates to -120. Once again, assume ClipU0 is considered first. Its value is incremented, and the branch function now evaluates to -121, which is further from our 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 PenU0 (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 ClipU0 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 Fergusons and Korel’s original rendering primarily involve simplifications to Ferguson and Korel’s, made possible by the spreadsheet evaluation model. These simplifications improve the efficiency of the approach, which is 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 node, CF-Chaining requires data-flow analysis to compute definitions of variables that may affect flow of control in that problem node. Such computation can be expensive, particularly in programs where aliasing must also be considered [24]. 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 [27] to update the visual display.

- CF-Chaining builds event sequences that encode lists of nodes that may influence a test case’s ability to execute a goal node; these sequences are constructed as problem nodes are encountered and used to guide attempts to solve constraints that affect the reachability of problem nodes. For 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 [27]).

- CF-Chaining’s algorithm is complicated by the presence of loops, which create semi-critical branches that may or may not prevent reaching a problem node. CF-Chaining uses a branch classification scheme to differentiate semi-critical branches, critical branches which necessarily affect reachability, and non-essential branches which require no special processing. Since our spreadsheets 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 which 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 case 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 is that no range information is available. In that case, our test case generation techniques consider all possible cell values within the default range of the data type.

A second scenario is that via a user’s help or a range information analysis tool, the test case generation techniques could obtain more precise knowledge of range information. With such (explicit) ranges, both techniques limit their search space to the specified ranges and generate test cases exactly within these ranges.
It seems plausible to think that availability and use of range information might affect the efficiency and effectiveness of test case generation techniques. Thus, our implementations of both the random and chaining techniques support both of the foregoing scenarios, and our empirical studies compare the techniques with and without range information.

4 Empirical Studies

Our test case generation methodology is intended to help users achieve du-a-adequate testing, which is communicated to the user with devices such as cell border colors. Determining whether this methodology achieves this goal requires user studies; however, before undertaking such studies we must first address more fundamental issues: namely, whether the methodology can successfully generate inputs to exercise du-associations, and whether it can do so efficiently. If the methodology is not sufficiently effective and efficient, there is no reason to pursue expensive studies involving human subjects.

Therefore, we begin by addressing the following research questions, the first concerning effectiveness, and the latter two concerning efficiency:

**RQ1:** Can our test case generation methodology generate test cases that execute a large proportion of the feasible du-associations of interest?

**RQ2:** How long will a user have to wait for our methodology to generate a test case?

**RQ3:** How long will a user have to wait for our methodology to report that it cannot generate a test case?

In addressing these questions we wish to also consider the differences between the Random and Chaining techniques, and whether provision of range information has any effect on results. Further, we wish to see whether differences in results occur across the three levels at which users can apply the methodology (spreadsheet, cell, and du-association).

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

4.1 Subjects

We used ten spreadsheets as subjects (see Table 3). 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 3 provides data indicating the complexity of the spreadsheets considered, including the numbers of cells, du-associations, expressions, and predicates contained in each spreadsheet.
<table>
<thead>
<tr>
<th>Spreadsheets</th>
<th>Cells</th>
<th>DU-assoc's</th>
<th>Feasible DU-assoc's</th>
<th>Expressions</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>25</td>
<td>56</td>
<td>90</td>
<td>53</td>
<td>10</td>
</tr>
<tr>
<td>Digits</td>
<td>7</td>
<td>89</td>
<td>61</td>
<td>35</td>
<td>14</td>
</tr>
<tr>
<td>FitMachine</td>
<td>9</td>
<td>121</td>
<td>101</td>
<td>33</td>
<td>12</td>
</tr>
<tr>
<td>Grades</td>
<td>13</td>
<td>81</td>
<td>79</td>
<td>42</td>
<td>12</td>
</tr>
<tr>
<td>MBTI</td>
<td>48</td>
<td>784</td>
<td>780</td>
<td>248</td>
<td>100</td>
</tr>
<tr>
<td>MicroGen</td>
<td>6</td>
<td>31</td>
<td>28</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>NetPay</td>
<td>9</td>
<td>24</td>
<td>20</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>NewClock</td>
<td>14</td>
<td>57</td>
<td>49</td>
<td>39</td>
<td>10</td>
</tr>
<tr>
<td>RandomJury</td>
<td>29</td>
<td>261</td>
<td>183</td>
<td>93</td>
<td>32</td>
</tr>
<tr>
<td>Solution</td>
<td>6</td>
<td>28</td>
<td>26</td>
<td>18</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3: Data about subject spreadsheets

Our test case generation prototype handles only integer type inputs; thus, all input cells in these subject spreadsheets are of integer type. Since commercial spreadsheets contain infeasible du-associations, all subject spreadsheets in our experiments also contain infeasible du-associations. To measure the effectiveness of our techniques at exercising feasible du-associations in this experiment, we determined all the infeasible du-associations through inspection.

4.2 Measures

Since our underlying testing system uses du-adequacy as a testing criterion, we measure a test case generation technique's effectiveness in terms of the percentage of feasible du-associations exercised by the test cases it generates.

As a measure a test case generation technique's efficiency we choose response time: the amount of wall clock time required to generate a test case 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, 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 case will increase. Thus, we gather these metrics incrementally over the course of repeated attempts to generate test cases for a spreadsheet, selected cell, or selected du-association arrow, automatically recording the new cumulative coverage reached whenever a new test case is generated, and recording the final level of coverage achieved, and 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 case generation techniques generate one test case at a time. However, the user may continue to invoke a technique to generate additional test cases. 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 case 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 case 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 repeatedly invoke our techniques and gather the required measurements. This approach raises several issues, as follows.

4.3.1 Automatic validation

The testing procedure under WYSIWYT is divided into two steps: finding a test case that executes one or more unexercised du-associations in the spreadsheet, and validating output cells as prompted. In these studies, because we are interested only in the test input generation step and do not have users performing validation, our scripts automatically validate all output cells whose validation causes some du-association in the area of interest (spreadsheet, selected cell, or du-association arrow) to be marked “exercised”. This approach simulates the effects of user validation under the assumption that, given a generated test case, the user validates all cells denoted as validatable in the area of interest following the application of that test case. Further, we do not measure validation time as part of our response time measurement.

4.3.2 Time limit

To simulate a user’s continued calls to test case generation techniques during incremental testing, our scripts repeatedly apply the techniques to the subject spreadsheet after each (automatic) validation of du-associations exercised by the preceding test case. 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 sufficient, long times in which our techniques can attempt to generate test cases.

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 case generation techniques can successfully generate test cases 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 cases.

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 cases 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 which users would be willing to wait for test case 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 cases 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 questions about the effects of input ranges, and our experiments investigate the use of techniques with and without range information. For cases where 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 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 case 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, test generation was employed to either attempt to generate test cases that exercise all du-associations in a spreadsheet, all du-associations in a cell or all du-associations associated with a du-association arrow.
In all three experiments, three independent variables were manipulated:

- The ten spreadsheets
- The test case 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 case generation techniques, with and without range information, starting from 35 sets of initial inputs. On each run, we recorded the times at which test case generation began, and the times at which untested du-associations were exercised or at which test case 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 non-input cells in that spreadsheet. As in the spreadsheet level experiment, we randomly assigned initial inputs. We then randomly chose a cell that was not fully tested by these initial inputs. The two test case 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 case generation began, and the times at which untested du-associations were exercised or at which test case generation failed; these measurements provided the values for our dependent variables. This process yielded (number-of-testable-cells-across-all-spreadsheets) × 3 × 2 × 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 inputs. The two test case generation techniques were then applied to the selected du-association arrow, with and without range information, under the given input assignment. On each run, we recorded the times at which test case generation began, and the times at which untested du-associations were exercised or at which test case generation failed; these measurements provided the values for our dependent variables. This setup yielded (number-of-feasible-du-associations-across-all-spreadsheets) × 2 × 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, and 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 case 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 address 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 case 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. Following that presentation, Section 4.7 discusses implications of these results.

4.6.1 Research Question 1: Ability to generate test cases

We turn first to our first research question: whether our test case generation methodology can generate test cases 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 cases 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. Both 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.

**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 4 lists, for each subject spreadsheet, the
ultimate effectiveness of Random and Chaining with and without range information, averaged across 35 runs.

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>RandomR</th>
<th>Chain</th>
<th>ChainR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>96.6%</td>
<td>100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Digits</td>
<td>59.4%</td>
<td>97.9%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>FitMachine</td>
<td>50.5%</td>
<td>50.5%</td>
<td>97.9%</td>
<td>97.9%</td>
</tr>
<tr>
<td>Grades</td>
<td>67.1%</td>
<td>99.8%</td>
<td>99.7%</td>
<td>99.9%</td>
</tr>
<tr>
<td>MBTI</td>
<td>25.6%</td>
<td>100.0%</td>
<td>99.9%</td>
<td>99.6%</td>
</tr>
<tr>
<td>MicroGen</td>
<td>71.4%</td>
<td>99.2%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>NetPay</td>
<td>40.0%</td>
<td>100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>NewClock</td>
<td>57.1%</td>
<td>100.0%</td>
<td>99.0%</td>
<td>99.4%</td>
</tr>
<tr>
<td>RandomJury</td>
<td>78.8%</td>
<td>83.2%</td>
<td>94.3%</td>
<td>92.7%</td>
</tr>
<tr>
<td>Solution</td>
<td>57.7%</td>
<td>78.8%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

Table 4: Ultimate effectiveness of techniques per spreadsheet.

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

We had expected that the addition of range information would improve the effectiveness of Chaining.
A comparison of the values in the two rightmost columns in Table 4, however, indicates that there was
little difference in 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, ad-
dition of range information actually decreased ultimate effectiveness, though by less than 2%. To determine
whether the differences in 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 (α < .05).4

Random without range information behaved much differently than Chaining, in only one case achieving an
ultimate effectiveness greater than 90% (Budget), and in six of ten cases achieving an 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 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 (α < .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 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

4 Statistical data was obtained using StatView 5.0.
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 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 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.)

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; this variance is an indicator of the degree of consistency that can be expected in technique effectiveness across applications.

Table 5 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. As the table shows, in most cases, the variance is small: on all but five spreadsheets less than 5%.

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>RandomR</th>
<th>Chain</th>
<th>ChainR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td>min</td>
<td>max</td>
</tr>
<tr>
<td>Budget</td>
<td>96.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>Digits</td>
<td>21.3</td>
<td>75.4</td>
<td>95.1</td>
<td>100.0</td>
</tr>
<tr>
<td>FitMachine</td>
<td>50.5</td>
<td>50.5</td>
<td>50.5</td>
<td>50.5</td>
</tr>
<tr>
<td>Grades</td>
<td>67.1</td>
<td>67.1</td>
<td>98.7</td>
<td>100.0</td>
</tr>
<tr>
<td>MBTI</td>
<td>25.6</td>
<td>25.6</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>MicroGen</td>
<td>71.4</td>
<td>71.4</td>
<td>92.9</td>
<td>100.0</td>
</tr>
<tr>
<td>NetPay</td>
<td>40.0</td>
<td>40.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>NewClock</td>
<td>57.1</td>
<td>57.1</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>RandomJury</td>
<td>78.7</td>
<td>79.8</td>
<td>79.8</td>
<td>86.9</td>
</tr>
<tr>
<td>Solution</td>
<td>57.7</td>
<td>57.7</td>
<td>57.7</td>
<td>80.8</td>
</tr>
</tbody>
</table>

Table 5: Variance in achieved coverage levels per technique per spreadsheet.

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 attained coverage level than we expected, did have variance equal to or larger than that for Random without ranges on all spreadsheets except Digits.

We also expected that Chaining would be more consistent than Random. This, however, was not the case on five spreadsheets FitMachine, (Grades, MBTI, NewClock, RandomJury) both with and without range information. However, variance exceeded 5% only on Grades and RandomJury for Chaining without ranges, and only on RandomJury for Chaining with range information.

We also expected range information to lower variance in coverage for Chaining, and this did occur on all spreadsheets save FitMachine, which had a coverage range of .5% more with range 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 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 medians and distribution in response times independent 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).

Successful response times, medians and distribution, spreadsheet level. Figure 8 presents boxplots showing all response data gathered in the spreadsheet level experiment, for Random and Chaining with and without ranges, per spreadsheet. Table 6 lists just the median response times shown in the plots.

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>RandomR</th>
<th>Chain</th>
<th>ChainR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>3.8</td>
<td>3.9</td>
<td>9.9</td>
<td>10.6</td>
</tr>
<tr>
<td>Digits</td>
<td>2.2</td>
<td>10.1</td>
<td>34.5</td>
<td>28.4</td>
</tr>
<tr>
<td>FitMachine</td>
<td>3.8</td>
<td>3.9</td>
<td>13.5</td>
<td>14.3</td>
</tr>
<tr>
<td>Grades</td>
<td>7.7</td>
<td>18.6</td>
<td>14.2</td>
<td>14.4</td>
</tr>
<tr>
<td>MBTI</td>
<td>37.5</td>
<td>40.0</td>
<td>31.2</td>
<td>30.7</td>
</tr>
<tr>
<td>MicroGen</td>
<td>2.7</td>
<td>8.1</td>
<td>6.4</td>
<td>6.5</td>
</tr>
<tr>
<td>NetPay</td>
<td>1.2</td>
<td>1.2</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>NewClock</td>
<td>2.3</td>
<td>2.5</td>
<td>10.4</td>
<td>9.3</td>
</tr>
<tr>
<td>RandomJury</td>
<td>80.2</td>
<td>28.7</td>
<td>182.3</td>
<td>173.2</td>
</tr>
<tr>
<td>Solution</td>
<td>1.3</td>
<td>1.4</td>
<td>18.9</td>
<td>17.9</td>
</tr>
</tbody>
</table>

Table 6: Median response times per spreadsheet, spreadsheet-level experiments

As the table shows, the median response times for Random with and without ranges were 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 with ranges were statistically significantly larger than response times for Random with ranges on all 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.
Figure 8: Response times (seconds) for experiments at the spreadsheet level.
Differences between Random with and without ranges, in contrast, were somewhat 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, 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.

**Successful response times versus coverage reached, spreadsheet level.** The preceding observation reveals a limitation of the two views of the data just considered: they 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 9 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 / range information combination, distinguished as indicated by the legend. Each point plotted represents the mean response time exhibited by the given technique / range information combination on all test cases which, when generated, resulted in a level of total coverage of the spreadsheet in the 10% range of coverage enclosing that point (as labelled 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 weakness 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 this limitation 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 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 achieves no coverage at higher coverage levels. This supports our earlier conjecture that the cases in which range information causes Random to attain higher median response times are due to its increased chances of success (albeit requiring more work) at covering du-associations not coverable in the absence of range information.

Comparing Chaining with Random, we see that Chaining has a smaller response time trend as coverage increases on six of the spreadsheet-level subjects (Digits, Grades, MBTI, MicroGen, RandomJury, and Solution). The remaining four spreadsheets show 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 improve to nearly equal to or better than the Random response times. Random
Figure 9: Response times (seconds) as coverage increases for experiments at the spreadsheet level.
reaches similar coverage levels and maintains lower response time trends than **Chaining** on only two spreadsheets (**Budget** and **NewClock**), but only with range information.

**Successful response times - differences among levels.** Our final view of response times concerns whether these times vary across the three levels of test generation: spreadsheet, individual cell, and individual du-association arrow. Figures 10 and 11, similar to Figure 8, present boxplots showing all response data gathered in the cell and du level experiments, respectively. Tables 7 and 8 list just the median times shown on the graphs.

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>RandomR</th>
<th>Chain</th>
<th>ChainR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Budget</strong></td>
<td>2.4</td>
<td>3.0</td>
<td>13.1</td>
<td>14.4</td>
</tr>
<tr>
<td><strong>Digits</strong></td>
<td>2.1</td>
<td>7.8</td>
<td>44.4</td>
<td>42.4</td>
</tr>
<tr>
<td><strong>FitMachine</strong></td>
<td>3.4</td>
<td>3.3</td>
<td>53.3</td>
<td>53.4</td>
</tr>
<tr>
<td><strong>Grades</strong></td>
<td>6.0</td>
<td>16.5</td>
<td>20.7</td>
<td>25.3</td>
</tr>
<tr>
<td><strong>MBTI</strong></td>
<td>13.9</td>
<td>26.4</td>
<td>20.1</td>
<td>20.0</td>
</tr>
<tr>
<td><strong>MicroGen</strong></td>
<td>3.8</td>
<td>3.8</td>
<td>11.5</td>
<td>11.4</td>
</tr>
<tr>
<td><strong>NetPay</strong></td>
<td>1.3</td>
<td>1.3</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td><strong>NewClock</strong></td>
<td>2.8</td>
<td>3.0</td>
<td>12.8</td>
<td>14.1</td>
</tr>
<tr>
<td><strong>RandomJury</strong></td>
<td>28.9</td>
<td>23.6</td>
<td>82.8</td>
<td>109.7</td>
</tr>
<tr>
<td><strong>Solution</strong></td>
<td>2.9</td>
<td>2.9</td>
<td>22.7</td>
<td>24.0</td>
</tr>
</tbody>
</table>

Table 7: Median response times per spreadsheet, cell-level experiments

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>RandomR</th>
<th>Chain</th>
<th>ChainR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Budget</strong></td>
<td>2.0</td>
<td>2.2</td>
<td>9.7</td>
<td>10.1</td>
</tr>
<tr>
<td><strong>Digits</strong></td>
<td>1.8</td>
<td>8.8</td>
<td>46.6</td>
<td>50.6</td>
</tr>
<tr>
<td><strong>FitMachine</strong></td>
<td>4.5</td>
<td>3.9</td>
<td>41.7</td>
<td>42.7</td>
</tr>
<tr>
<td><strong>Grades</strong></td>
<td>6.6</td>
<td>11.3</td>
<td>27.1</td>
<td>25.6</td>
</tr>
<tr>
<td><strong>MBTI</strong></td>
<td>7.0</td>
<td>76.7</td>
<td>31.3</td>
<td>28.7</td>
</tr>
<tr>
<td><strong>MicroGen</strong></td>
<td>2.6</td>
<td>2.7</td>
<td>8.0</td>
<td>8.5</td>
</tr>
<tr>
<td><strong>NetPay</strong></td>
<td>0.5</td>
<td>0.9</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td><strong>NewClock</strong></td>
<td>1.8</td>
<td>2.1</td>
<td>8.3</td>
<td>8.3</td>
</tr>
<tr>
<td><strong>RandomJury</strong></td>
<td>39.9</td>
<td>22.6</td>
<td>86.6</td>
<td>91.1</td>
</tr>
<tr>
<td><strong>Solution</strong></td>
<td>1.2</td>
<td>2.0</td>
<td>21.4</td>
<td>22.3</td>
</tr>
</tbody>
</table>

Table 8: Median response times per spreadsheet, du-level experiments

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 less 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 where response times were already low at
Figure 10: Response times (seconds) for experiments at the cell level.
Figure 11: Response times (seconds) for experiments at the du level.
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, whose 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 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, by definition, hardcoded as a constant. Thus, we focus on Chaining.

Figure 12 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. Table 9 lists median response times.

<table>
<thead>
<tr>
<th></th>
<th>Spreadsheet</th>
<th>Cell</th>
<th>Du-Assoc. Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Chain</td>
<td>ChainR</td>
<td>Chain</td>
</tr>
<tr>
<td>Budget</td>
<td>61.6</td>
<td>59.3</td>
<td>30.1</td>
</tr>
<tr>
<td>Digits</td>
<td>1612.6</td>
<td>2550.1</td>
<td>704.2</td>
</tr>
<tr>
<td>FitMachine</td>
<td>612.5</td>
<td>226.3</td>
<td>0.4</td>
</tr>
<tr>
<td>Grades</td>
<td>48.3</td>
<td>48.9</td>
<td>0.9</td>
</tr>
<tr>
<td>MBTI</td>
<td>34.4</td>
<td>46.2</td>
<td>81.2</td>
</tr>
<tr>
<td>MicroGen</td>
<td>11.8</td>
<td>11.8</td>
<td>17.2</td>
</tr>
<tr>
<td>NetPay</td>
<td>9.9</td>
<td>9.3</td>
<td>6.7</td>
</tr>
<tr>
<td>NewClock</td>
<td>86.4</td>
<td>89.2</td>
<td>27.0</td>
</tr>
<tr>
<td>RandomJury</td>
<td>1303.0</td>
<td>15.1</td>
<td>201.0</td>
</tr>
<tr>
<td>Solution</td>
<td>12.1</td>
<td>11.9</td>
<td>17.5</td>
</tr>
</tbody>
</table>

Table 9: Median response times on failure for Chaining at the spreadsheet, cell, and du-association arrow levels. The notation “—” indicates that no failures occurred.

As the table shows, median response times on failure vary widely across both spreadsheets and level of application. The only clear trend that emerges from the data involves the fact that the use of range information only infrequently results in differences in failure response times. Median times differ 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 requires 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 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).
Figure 12: Response times (seconds) for failing runs of Chaining, with and without ranges, at all levels of application.
Of greater interest is the relationship between failure response times and the success response times listed in Table 6. In almost all cases median response time for failure was higher than median response time for success (sometimes nearly 100 times higher). (Exceptions to this 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 tradeoffs 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 case generation for spreadsheets is feasible. In the cases we considered, Chaining was highly effective (both with and without range information) at generating test cases, 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 now 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 case generation might perform nearly as well as more complex heuristics, thus providing a more easily implemented approach to test case 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 had 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 case 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, in this application, necessarily terminates, 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 cases for more focused areas of a spreadsheet (individual cells or du associations), test case generation techniques could respond more quickly than when invoked with the entire spreadsheet as target. Comparing our results on these three levels, however, 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, but supporting such a suggestion requires study of users. In any case, the results suggest that all three levels of test generation may have utility.

5 Conclusions and Future Work

We have presented an automated test case generation methodology for spreadsheet languages. Our methodology uses an incremental generation strategy, and is driven by the end user’s request, allowing the user to generate test cases 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 case generation techniques within our methodology, one based on random generation and the second based on Ferguson and Kored’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 cases.

Our results support several suggestions about the use of automated test case 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 provision 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, would 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.

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 experiments involving human users of our methodology. The results presented in this paper motivate the choice, for that experimentation, of either the Chaining technique for test case 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.

Acknowledgments

We thank the Visual Programming Research Group for their work on Forms/3 and their feedback on our methodologies. This work has been supported by the National Science Foundation by ESS Award CCR-9806821 and ITR Award CCR-0082265 to Oregon State University.

References


