Automated Test Case Generation for Spreadsheets

Marc Fisher, Mingming Cao, Gregg Rothermel, Curtis R. Cook, Margaret M. Burnett

Computer Science Department
Oregon State University
Corvallis, Oregon
grother@cs.orst.edu

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 automate test case generation for spreadsheets 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 one involving a goal-oriented approach. We describe these techniques, and report results of an experiment examining their relative costs and benefits.

1. INTRODUCTION

Perhaps the most widely used programming paradigm today is the spreadsheet paradigm. Little research, however, has addressed the software engineering tasks that arise in creating and maintaining spreadsheets. This inattention is surprising given the role played by spreadsheets on significant matters such as budgets, grades, and business decisions.

In fact, recent research reports that spreadsheets often contain faults. A survey of the literature [20] provides details: in four field audits of operational spreadsheets, faults were found in an average of 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, 29].

To help address these problems, in previous work we presented a methodology for testing spreadsheet formulas [22, 24]. Our "What You See Is What You Test" (WYSIWYT) methodology lets spreadsheet users incrementally apply test inputs and validate outputs, and provides visual feedback about the effectiveness of their testing. Empirical studies have shown that this methodology can help users test their spreadsheets more adequately and more efficiently [25].

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., [6, 8, 10, 13, 14, 15, 21]), 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 [10]. We describe these techniques and their integration into our WYSIWYT methodology, and report results of experiments examining their efficiency and effectiveness.

2. BACKGROUND

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, 5, 16, 27]) based on this paradigm.

In this article, we present examples of spreadsheets in the research language Forms/3 [4]. Figure 1 shows an example of a Forms/3 spreadsheet, `GrossPay`, which calculates an employee's weekly pay given a payrate and the hours they worked during that week. As the figure shows, Forms/3 spreadsheets, like traditional spreadsheets, consist of cells; however, these cells are not restricted to grids. Also, in the figure cell formulas are displayed, but in general the user can display or hide formulas.

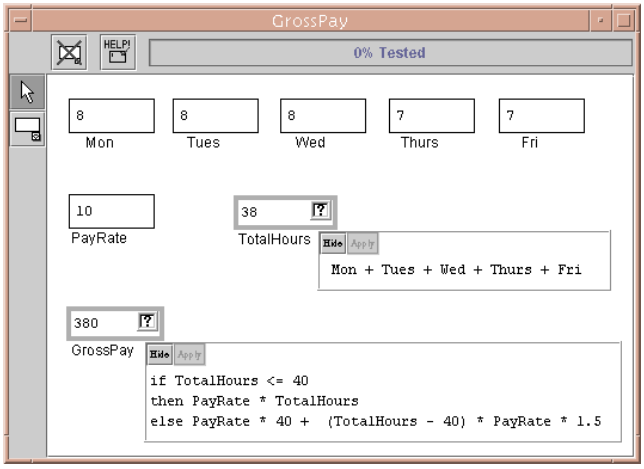

Figure 1: Forms/3 spreadsheet GrossPay.

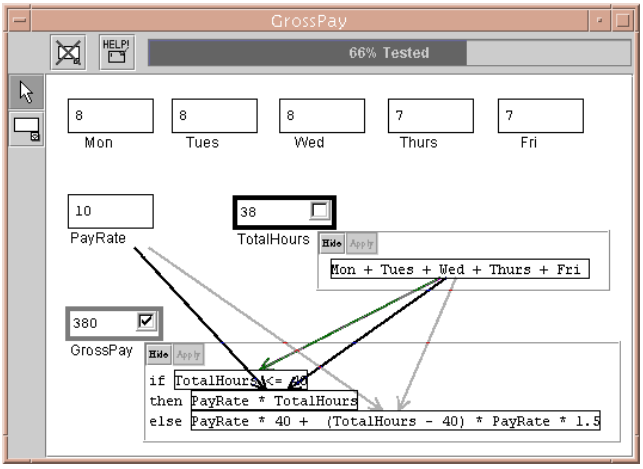

Figure 2: Forms/3 spreadsheet GrossPay with testing information displayed after a user validation.

2.1 The WYSIWYT Methodology

In our "What You See Is What You Test" (WYSIWYT), methodology for testing spreadsheets [22, 24, 25], 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 `GrossPay` spreadsheet by entering cells and formulas, reaching the state shown in Figure 1. 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 aren't testable.)

Suppose the user looks at the values displayed on the screen and decides that cell `GrossPay` 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 2, 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.)

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 `GrossPay`, references in the `then` clause have now been exercised, but references in the `else` clause have not; thus, that cell's border is partially blue (dark 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 at references in the formula for `TotalHours` have been exercised; hence, that cell's border is now fully blue (black 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 ending at cell `GrossPay`. These arrows depict testedness information at a finer granularity, following the same color scheme as for the cell borders.

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 [9]; for brevity we call our adaptation of this criterion the *du-adequacy* criterion. We precisely define this criterion in [22]; here, we summarize that presentation.

The du adequacy criterion is defined through an abstract model of spreadsheets called a *cell relation graph* (CRG). Figure 3 shows the CRG for spreadsheet GrossPay. 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

[1] Other criteria are discussed in [23].

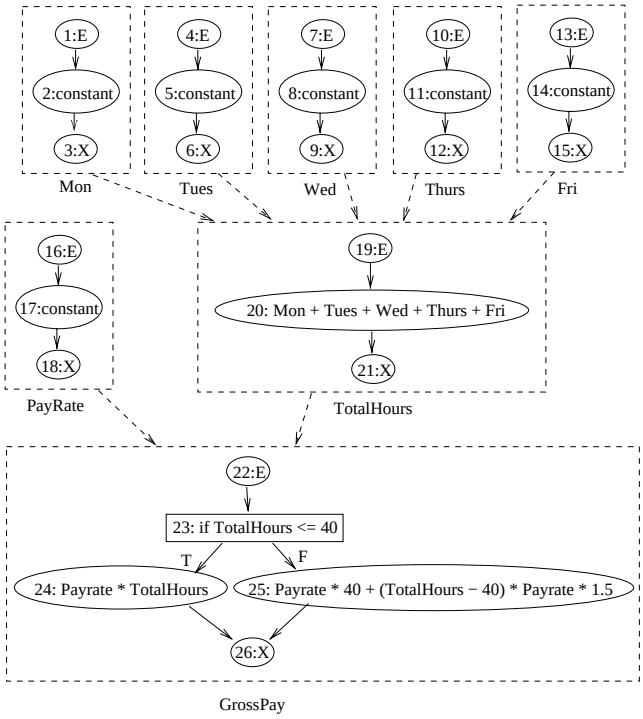

Figure 3: Cell relation graph for GrossPay

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 [12, 28].

2.2 ATCG techniques for imperative languages

There has been much research on techniques for automatic test case generation (ATCG) for imperative languages. Ferguson and Korel [10] classify test case generation techniques according to their mechanism of generation into three categories: *random*, *path-oriented* and *goal-oriented*.

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 requirement, and then attempt to find input values that cause that path to be executed. Ferguson and Korel distinguish two types of path-oriented techniques: those based on *symbolic execution*, and those that are *execution-oriented*. Techniques based on symbolic execution [6, 8, 13, 18] use symbolic execution to find the constraints, in terms of input variables, that must be satisfied in order to execute a target path, and attempt to solve this system of constraints. The solution provides a test case for that path. A disadvantage of these techniques is that they can waste effort attempting to find inputs for infeasible paths; they can also require large amounts of memory to store the expressions encountered during symbolic execution, and powerful constraint solvers to solve complex equalities and inequalities. They may also have difficulties handling complex expressions. Execution-oriented techniques [15] 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 [10, 14], 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 [10].

3. AUTOMATED TEST CASE GENERATION FOR SPREADSHEETS AND WYSIWYT

In any testing methodology for end users that does not automatically generate test cases, the users themselves must generate useful test inputs, and this can be difficult. To help with this, we have developed a test case generation methodology for spreadsheets, integrated support for that methodology with 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 a basic version of that methodology. Sections 3.2 and 3.3 then present refinements.

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 arrows on the visible display (or, selecting none, express 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, and its first task is to determine the set $UsefulDUs$ of du-associations relevant to the user's request (du-associations in the area of interest that have not been validated). If the user has not indicated specific cells or 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 arrows in the spreadsheet, $UsefulDUs$ includes each of the unvalidated du-associations associated with each such arrow.

The second task of the system is 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 [22]), it is available in data structures kept by the engine, and is retrieved from there.

Given $UsefulDUs$ and $InputCells$, the test case generation system can attempt to generate a test case. The system saves the existing configuration of input cell values for restoration if generation fails, and then invokes a test case generation technique.

As Section 2.2 indicates, there are many test generation techniques that could be utilized; the simplest of these is to randomly generate input values. We implemented this technique in our prototype; we refer to it as `Random`. (We use `Random` here to illustrate our overall methodology; Section 3.2 describes a second technique.)

`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 execute 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, 7].

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 already 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 2.1). 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 `GrossPay` in Figure 2, the system would manipulate the values in cells `Mon`, `Tues`, etc., and would present only `GrossPay` as the relevant validatable cell.

If, on the other hand, the system reaches a built-in time limit without `Random` finding a useful set of test inputs, it restores the previous input state and tells the user that it has been unable to generate a test case. Whether or not the system has succeeded in generating a test case, the user at this point can validate an output value, or can ignore the generated values (such as due to dislike of the input set generated) and choose to try again with the same or different cells or arrows selected, in which case the test case generation technique attempts again using 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.[2] 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" [10] for our purpose; we call our adaptation `Chaining`.

Like `Random`, `Chaining` is invoked, in our methodology, to find a set of inputs that exercise one or more du-associations in $UsefulDUs$. Unlike `Random`, `Chaining` accomplishes its task by iterating through $UsefulDUs$, considering each du-association in turn. (In contrast, `Random` simply generates inputs for all cells in $InputCells$, and then checks whether any du-association in $UsefulDUs$ is exercised.) On finding such a set, `Chaining` terminates, and the visual devices described above for indicating relevant validatable output cells are activated. If `Chaining` fails on all du-associations in $UsefulDUs$, then like `Random`, it indicates this to the system, which reports that it could not find a test case.

We now describe the process by which, in considering a du-association (d,u), `Chaining` proceeds.[3] 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.[4] For example, to exercise du-association (20,25) in `GrossPay` (see the CRG in Figure 3 and its associated spreadsheet in Figure 2), input values must cause node 20 in the formula graph for `TotalHours` to be reached (any set of input values achieves this), and they must also cause node 25 in the formula graph for cell `GrossPay` to be reached.

The conditions that must most immediately be met 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, any predicate nodes lying on the direct path from e to d (or u), and d

[2] Personal communication, Jeff Offutt.

[3] Due to space limitations the description is somewhat abbreviated; a more detailed version, with additional description of Ferguson and Korel's approach, can be found in [11].

[4] An 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 [22].

(or u). For example, the constraint path for nodes 20 and 25 in the CRG for `GrossPay` are (19,20) and (22,23,25), 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 (20,25) in `GrossPay` is (19,20,22,23,25).

When considering du-association (d, u), `Chaining` first constructs the constraint path for (d, u). Given our methodology, it is necessarily the case that under current inputs, (d, u) is not exercised – if (d, u) were exercised, it would not be included in $UsefulDUs$. Thus, it must be the case that under 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 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. In the example we have been considering, the relevant concatenated execution trace, assuming the spreadsheet's input cells have values as shown in Figure 2, is (19,20,22,23,24). `Chaining` identifies the *break point* in the constraint path: the pair of nodes consisting of the first node in the concatenated execution traces not present in the constraint path, together with the predicate node immediately preceding it (or, less formally, the earliest "incorrectly taken branch" in the execution traces). In our example, the break point is (23,24).

Given a break point (n_1,n_2), `Chaining`'s next task is to find inputs that cause the predicate in n_1 to take the opposite branch. To do this, the technique uses a constrained linear search procedure over the input space; we describe this procedure later in this section. Three outcomes of this search procedure are possible. (1) The search does not succeed. In this case, n_1 is designated a *problem node* and dealt with by a procedure described momentarily. (2) The search succeeds, and du-association (d,u) is now executed. In this case, the technique has succeeded and terminates. (3) The search succeeds, and inputs that cause the desired branch from the predicate in n_1 have been found, but a subsequent predicate on the constraint path has not been satisfied (i.e., another break point exists), and (d,u) has not yet been executed. In this case, `Chaining` repeats the above process, finding the next break point and initiating a new search, to try to make further progress.[5]

In the example we have been considering, the only outcomes possible are outcomes 1 and 2: the search fails, or it succeeds causing du-association (20,25) to be exercised. If, however, cell `GrossPay` had contained another predicate node p in between nodes 23 and 25, such that node 25 is reached only if p evaluates to "true", then outcome 3 could have occurred, e.g., if the inputs found to cause 23 to evaluate to "false" did not also cause p to evaluate to "true".

When a problem node is encountered, `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, `Chaining` will be able to make progress. Thus, faced with a problem node n_1, `Chaining` collects a set $ChainDUs$ of other du-associations (d',u') in the spreadsheet that have two properties: (1) u' is the predicate use associated with the alternate branch of n_1, i.e. the branch we wish to take (recall from Section 2.1 that a predicate use is an out-edge of a predicate node, and represents either the "true" or "false" outcome of that predicate); (2) d' is not currently exercised. These du-associations, if exercised, necessarily enable the desired branch to be taken. `Chaining` iterates through du-associations in $ChainDUs$, applying (recursively) the same process described for use on (d,u) to each.[6]

The Search Procedure

The search procedure used by `Chaining` to find inputs that cause predicates to take alternative branches involves two steps. First, a *branch function* 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 turn.

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 [10], 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 combine other branch functions in a meaningful way. (The functions are further described in [11].)

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 [10], a sequence of "exploratory" and "pattern" moves

[5] 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 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 being iterated on.

[6] As discussed in [10], 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 implementation.

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

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

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

Table 2: Branch functions for logical operators.

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. This search procedure is described in detail in [10] and we refer the reader there for further description.

Comparison of Approaches

Space does not permit a detailed comparison of Ferguson and Korel's [10] `Chaining` technique and our adaptation of that technique. For readers familiar with that approach, however, we summarize the differences between approaches.

For brevity in the following, we refer to Ferguson and Korel's technique as `CF-Chaining`, due to 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 [19]. 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 [22] 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 [22]).
- `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.

These differences primarily involve simplifications to Ferguson and Korel's approach, made possible by the spreadsheet evaluation model. As such, this work illustrates the potential suitability of Ferguson and Korel's overall approach to that evaluation model. These simplifications improve the efficiency of the approach, which is a critical matter in spreadsheets, which feature rapid response.

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 possible 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.

We believed 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 supported both of these scenarios, and our empirical studies investigated the techniques with and without range information.

4. EMPIRICAL STUDIES

Our test case generation methodology is intended to help users achieve du-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 questions: namely, whether the methodology can in fact generate inputs that exercise

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

Table 3: Data about subject spreadsheets

a sufficient number of feasible du-associations, and whether it can do so sufficiently efficiently. If the answers to these questions are negative, there is no reason to pursue studies involving human subjects. We must also determine whether either `Random` or `Chaining` is more efficient or effective than the other, and whether provision of range information is necessary. Therefore, in our initial empirical studies, we focus on these fundamental questions:

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

RQ2: How do `Random` and `Chaining` compare, within our methodology, in terms of effectiveness and efficiency?

RQ3: Does the provision of range information alter the effectiveness and efficiency of `Random` and `Chaining`?

To investigate these questions, 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. In the experiments reported here, we focus on test case generation at the whole spreadsheet level.

4.1 Subjects

We used ten spreadsheets as subjects (see Table 3). These spreadsheets were 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.

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

To investigate our research questions we use two measures: *effectiveness* and *efficiency*. Since our underlying testing system uses du-adequacy as a testing criterion, we measured a test case generation technique's effectiveness by the percentage of feasible du-associations exercised by the test cases it generated. To measure a test case generation technique's efficiency, we measured the amount of (wall clock) time required to generate a test case that exercises one or more du-associations.

4.3 Experiment Methodology

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 (and we expect will) 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 execute, and the time required to generate a new useful test case will increase. We wish to consider differences in efficiency across this process. Further, it is only through a process of 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 repetitively to a spreadsheet 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, since 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 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 validatable cells for that test case. We do not measure validation time as part of our efficiency measurement.

4.3.2 Time limit

To simulate a user's repetitive 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 efficiency more generally and discover what sort of internal time limits might be appropriate. Thus, our scripts must provide sufficient time for our techniques to attempt to generate test cases, and time out when a limit is reached.

To determine what time limits to use, we performed several trial runs with extremely long limits (several hours) per script. We then determined the time after which (in these runs) no additional test cases were found, and used this to set our limits. We chose a time limit of 1200 seconds for all spreadsheets other than MBTI and RandomJury; for these

spreadsheets we chose time limits of 10000 and 5000 seconds, respectively.[7]

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 efficiency 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 every feasible du-association in each of our subject spreadsheets.

To investigate the use of ranges, we needed to provide reasonable ranges, such as could 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 might be accomplished by the user, or by the test generation mechanism itself.)

4.3.5 Initial values

Another consideration that might affect the effectiveness and efficiency of our 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, whereas `Chaining` starts from the current values of input cells and searches the input space under the guidance of branch functions until it finds a solution. Thus, `Random` is independent of initial values whereas initial values could affect `Chaining`. To control for the effects of initial values, and allow fair comparisons of our techniques, 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 apply runs of techniques in pairs, with each pairing starting from the same sets of initial values.

4.4 Experiment Design

Our experiment evaluated our automatic test case generation methodology on ten subject spreadsheets at the whole spreadsheet level. The three independent variables manipulated in this experiment are:

- The ten spreadsheets
- The test case generation technique
- The use of range information

We measured two dependent variables:

- effectiveness
- efficiency

The experiment employed a $10 \times 2 \times 2$ factorial design with 35 different initial input configurations per spreadsheet. For each subject spreadsheet, we applied each of our two test case generation techniques starting from 35 sets of initial inputs without range information. We then did the same using ranges. On each run, we measured the times at which untested du-associations were exercised; these measurements provided the values for our dependent variables. These runs yielded 1400 sets of effectiveness and efficiency measurements for our analysis. All runs were conducted, and all timing data collected, on a Sun Microsystems Ultra 10 with 128 MB of memory.

4.5 Results and Analysis

We now present our results and analysis, first considering effectiveness, and then considering efficiency.

4.5.1 Effectiveness

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 in the subject spreadsheets – we refer to this as their *ultimate effectiveness*. Table 4 lists, for each of the subject spreadsheets, the ultimate effectiveness of `Random` and `Chaining` with and without range information, averaged across 35 runs.

	`Random` without Range	`Random` with Range	`Chaining` without Range	`Chaining` with Range
Digits	59.44%	97.89%	100.00%	100.00%
FitMachine	50.50%	50.50%	97.93%	97.90%
Grades	67.10%	99.82%	99.71%	99.89%
MBTI	25.64%	100.00%	99.87%	99.64%
MicroGen	71.43%	99.18%	100.00%	100.00%
NetPay	40.00%	100.00%	100.00%	100.00%
NewClock	57.14%	100.00%	99.01%	99.36%
Budget	96.57%	100.00%	100.00%	100.00%
RandomJury	78.78%	83.23%	94.29%	92.69%
Solution	57.69%	78.79%	100.00%	100.00%

Table 4: Ultimate effectiveness of techniques per spreadsheet, with and without explicit range information, on average over 35 runs.

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`. However, comparing the values in the two rightmost columns in Table 4 indicates that there was little difference in ultimate effectiveness between `Chaining` with and without range information. In fact, on `FitMachine` and `RandomJury`, the two

[7] Obviously, we cannot guarantee that longer time limits would not allow the techniques to exercise additional du-associations. However, our limits likely exceed the time which users would be willing to wait for test case generation to succeed, and thus for practical purposes are sufficient.

cases in which there was the greatest potential for improvement, addition of range information actually decreased (by less than 2%) ultimate effectiveness. To determine whether the differences in ultimate effectiveness between `Chaining` with and without range information were statistically significant, we used unpaired t-tests on pairs of effectiveness values per technique per spreadsheet. The differences between the techniques were statistically significant only for `MBTI` ($\alpha < .05$).[8]

`Random` without range information behaved much differently than `Chaining`. In only one case did `Random` without range information achieve an ultimate effectiveness greater than 90% (`Budget`), and in six of ten cases it achieved an ultimate effectiveness less than 60%. Ultimate effectiveness also varied widely for this technique, ranging from 25.64% to 96.57%. 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%.) Unpaired t-tests showed that the effectiveness differences between `Random` without ranges and `Chaining` without ranges were all statistically significant ($\alpha < .05$).

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

Addition of range information to `Random` also helped its performance in comparison to `Chaining`. On two spreadsheets, `MBTI` and `NewClock`, `Random` with range information achieved greater 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.)

Our second view of effectiveness (Table 5) considers the ability of the four test case generation techniques, across all ten subject spreadsheets, to generate test cases reaching four different levels of effectiveness. The table shows the number of times, out of 350 runs, each technique achieved at least 50%, 75%, 90%, or 100% effectiveness. As the table shows, `Random` without range information is the only technique to sometimes fail to reach 50% effectiveness, and is far less successful than the other techniques at reaching higher levels of effectiveness. Also, `Chaining` techniques achieved 75%, 90%, and 100% effectiveness more often than `Random` with range information, and `Chaining` without range information achieved 90% and 100% effectiveness a few more times than does `Chaining` with range information.

4.5.2 *Efficiency*

We also consider two views of efficiency. First, Table 6 reports response time characteristics of the test case generation techniques; that is, the amount of time the user must wait after pushing the "Help Me Test" button until a suitable test case is displayed. The table shows, for each technique, the number of total test cases successfully generated by that technique, and the numbers and percentages of the times in which test case generation succeeded in less than 4 seconds, and less than 10 seconds, respectively.[9]

As the table illustrates, with all techniques, a test case was generated within 4 seconds over 70% of the time, and within 10 seconds over 84% of the time. With or without range information, `Chaining` achieved faster response times more often than `Random`. The table also shows that `Random` without ranges was more responsive than `Random` with ranges at the 4 second mark; however, this should be qualified by the fact that `Random` without ranges was able to generate only about one-eighth as many test cases as `Random` with ranges. Finally, the table shows that the use of range information did improve `Chaining`'s response time: in fact, `Chaining` with range information achieved the greatest number of low response times, succeeding within 4 seconds on over 90% of the runs, and within 10 seconds on over 95% of the runs.

Table 7 provides a second view, showing the efficiencies of techniques relative to various levels of coverage across all ten subject spreadsheets as the techniques are applied repeatedly; that is, how long test generation might take to attain the desired level of coverage. For each of the four techniques, four levels of coverage (50%, 75%, 90%, and 100%) are considered.

As the table shows, both `Chaining` techniques achieved 100% coverage much more quickly (over 400 seconds more quickly) than their `Random` counterparts. `Chaining` with range information was fastest, reaching each level of coverage more quickly than `Chaining` without range information; however, the time differences between the two `Chaining` techniques never exceeded 94 seconds. Further, `Chaining` without range information was somewhat slower than `Random` with range information at reaching the 50%, 75%, and 90% levels of coverage, although the difference never exceeded 54 seconds, and decreased as coverage level increased.

Two entries in this table bear further scrutiny. `Random` without range information appears surprisingly fast at the 90% coverage level; however, this should be interpreted in light of the small number of cases in which the technique

	50%	75%	90%	100%
`Random` without Range	274	76	35	5
`Random` with Range	350	312	245	218
`Chaining` without Range	350	349	347	252
`Chaining` with Range	350	350	342	239

Table 5: Number of runs (out of 350) in which techniques exceeded specific levels of effectiveness.

	Total test cases generated	Test cases generated in < 4 secs	Test cases generated in < 10 secs
`Random` w/o Range	1318	1026 (77.8%)	1114 (84.5%)
`Random` w Range	9731	6886 (70.8%)	8239 (84.7%)
`Chaining` w/o Range	12750	10354 (81.2%)	11262 (88.3%)
`Chaining` w Range	12115	11001 (90.8%)	11605 (95.8%)

Table 6: Number and percentage of test cases generated in less than 4 and less than 10 seconds, across all 10 spreadsheets.

[8] Statistical data was obtained using StatView 5.0; further details on the analysis technique applied and the data obtained are available in [11].

[9] In the literature on usability, 4 seconds is cited as an appropriate limit for common tasks [26], and 10 seconds as a limit for keeping a user's attention focused on a task [17].

	50%	75%	90%	100%
Random without Range	66.3	79.9	13.0	622.4
Random with Range	9.9	62.9	93.8	521.1
Chaining without Range	63.9	102.3	130.7	119.4
Chaining with Range	15.2	27.7	45.7	25.9

Table 7: Average cumulative time (seconds) required to achieve given levels of coverage, considering only test cases that reached those levels (see numbers reported in Table 5).

achieved that level of coverage (see Table 5). Specifically, the technique reached the 90% coverage level on only 35 test cases (all on one spreadsheet), so these results simply show that on that one spreadsheet, 90% coverage could be achieved quickly. Similarly, results for Random without range information at the 100% coverage level are based on only the five test cases on which the technique reached that level.

4.6 Threats to Validity

This experiment, like any other, has limitations (threats to validity) that must be considered when assessing its 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. Finally, 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. Also, to support fair timing comparisons of our techniques, our implementations of techniques shared code wherever possible, differing only where required by the underlying algorithms.

Finally, 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.

4.7 Discussion

Keeping in mind the limitations imposed by the threats to validity just described, our results have several implications.

First, our results suggest that, from the point of view of effectiveness and efficiency, automated test case generation for spreadsheets seems to be 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. In a high percentage of cases, Chaining achieved this coverage within reasonable time limits. These results thus motivate further work on test case generation techniques for spreadsheets, and on the Chaining technique, and suggest that studies of whether end users can profit from the use of such techniques would be appropriate.

Our results also highlight several tradeoffs between techniques. First, we had initially conjectured that with spreadsheets, random test case generation might perform nearly as well as a more complex heuristic, thus providing a more easily implemented approach to test case generation. Our experiments suggest that this conjecture is false. In the cases we observed, 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 larger than that of Chaining techniques. Random techniques also exhibited fast response times less often than Chaining techniques, and although a Random technique using range information was capable of achieving low levels of coverage more quickly than Chaining without range information, the latter outperformed the former at high levels of coverage.

At the outset of this work we also postulated that provision of range information would benefit both test case generation techniques. Where Random was concerned, this proved correct: Random with ranges often achieved far greater levels of coverage than Random without ranges. We were surprised, however, that Chaining did not benefit, in terms of effectiveness, from the provision of range information. In fact, Chaining without range information was marginally better than Chaining with range information at achieving higher coverage levels. Also, while range information did help Chaining achieve results more quickly than Chaining without such information, the speedup was not large. 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, though when it does find solutions it can do so in fewer steps.

Overall, these results support some stronger suggestions about automated test case generation for spreadsheets:

- Given a choice, one should implement Chaining rather than Random.
- If one can implement only Random, one should make provision for providing range information.
- If one chooses to implement Chaining, provision of range information might marginally harm effectiveness, but might marginally improve efficiency.

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 on a dynamic, goal-oriented approach. The details of these techniques, however, do not need to be known by the end users of our system. We have prototyped our methodology, and our empirical studies suggest that it can effectively and efficiently generate test cases.

Given these results, our next step in this research is the design and performance of additional experiments, including (1) experiments with a wider range of spreadsheets, including representatives of commercial spreadsheet applications, (2) experiments using our methodology at the individual du-pair and individual cell levels, and (3) experiments involving human users of our methodology. The results presented in this paper motivate the choice, for that experimentation, of the `Chaining` technique for test case generation, with 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.

6. REFERENCES

[1] N. Belkin. Helping people find what they don't know. *Communications of the ACM*, 41(8):58–61, Aug. 2000.

[2] D. Bird and C. Munoz. Automatic generation of random self-checking test cases. *IBM System Journal*, 22(3):229–245, 1983.

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

[4] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, 2001.

[5] E. H. Chi, P. Barry, J. Riedl, and J. Konstan. A spreadsheet approach to information visualization. In *IEEE Symposium on Information Visualization*, Oct. 1997.

[6] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, Sept. 1976.

[7] C. Corritore, B. Kracher, and S. Wiedenbeck. Trust in the online environment. In *Proceedings of HCI International*, pages 1548–1552, Aug. 2001.

[8] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, Sept. 1991.

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

[10] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, Jan. 1996.

[11] M. Fisher, M. Cao, D. Brown, G. Rothermel, C. R. Cook, and M. M. Burnett. Integrating automated test case generation into the WYSIWYT spreadsheet testing methodology. Technical Report TR: 02-60-01, Oregon State University, Jan. 2002.

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

[13] A. Gotlieb, B. Botella, and M. Reuher. Automatic test data generation using constraint solving techniques. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 53–62, Mar. 1998.

[14] B. Korel. A dynamic approach of automated test data generation. In *Proceedings of the International Conference on Software Maintenance*, pages 311–317, Nov. 1990.

[15] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–897, Aug. 1990.

[16] J. Leopold and A. Ambler. Keyboardless visual programming using voice, handwriting, and gesture. In *Proceedings of the 1997 IEEE Symposium of Visual Languages*, pages 28–35, Sept. 1997.

[17] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, CA, 1994.

[18] A. J. Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, Nov. 1991.

[19] H. Pande, W. Landi, and B. Ryder. Interprocedural def-use associations in C programs. *IEEE Transaction on Software Engineering*, 20(5):385–403, May 1994.

[20] R. Panko. What we know about spreadsheet errors. *Journal of End User Computing*, pages 15–21, Spring 1998.

[21] C. Ramamoorthy, S. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, Dec. 1976.

[22] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering*, pages 110–147, Jan. 2001.

[23] G. Rothermel, L. Li, and M. Burnett. Testing strategies for form-based visual programs. In *Proceedings of the 8th International Symposium on Software Reliability Engineering*, pages 96–107, Nov. 1997.

[24] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 20th International Conference on Software Engineering*, pages 198–207, Apr. 1998.

[25] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[26] B. Shneiderman. *Designing the User Interface*. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[27] T. Smedley, P. Cox, and S. Byrne. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *Advanced Visual Interfaces '96*, May 1996.

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

[29] E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, and C. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *ACM CHI'97*, pages 22–27, Mar. 1997.