Previous work has developed the What You See Is What You Test (WYSIWYT) methodology for testing spreadsheets. This methodology has been shown to help end users test, debug, and modify spreadsheets. To date, however, this system has provided no support for creating, reusing, and managing test cases, a process that can be tedious and time-consuming. To alleviate this, we have developed automated test case generation and test reuse methodologies for the WYSIWYT methodology. For test case generation, we have prototyped two techniques, and performed a study to assess the effectiveness and efficiency of these techniques. The results of this study show that we can efficiently exercise a large percentage of a spreadsheet under test. We also implemented a test reuse methodology and performed a study that shows that we are able to find a relatively small subset of test cases to reuse after a modification to a spreadsheet, and that these test cases re-establish much of the coverage lost by modifications made to the spreadsheet.
Helping End Users Create and Manage Test Cases in the WYSIWYT Methodology

by

Marc Randall Fisher II

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed August 23, 2002
Commencement June 2003
I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Marc Randall Fisher II, Author
ACKNOWLEDGMENTS

I would like to thank Dr. Rothermel for all of his help on this thesis. Without his invaluable assistance and support this thesis would not have been completed.

My thanks to Dr. Burnett for developing Forms/3, the vehicle for this research, and for her help throughout the development of this research. Also to Dr. Cook for his insights at various times during the research, and to Dr. Stetz for being on my committee.

I am grateful for the assistance of the various members of the Forms/3 group throughout the development of this research, in particular to Mingming Cao and Dalai Jin, who started much of the work that this built on, Andy Ko, who developed the larger spreadsheets used in the study, Dan Keller, Josh Cantrell, Aaron Wilson, and Eugene Creswick, who developed and enhanced the user interface for Help Me Test.

Additionally I am particularly appreciative of the help of Darren Brown, whose work on the test generation experiment was invaluable.

Finally, I must thank my wife, Tamara, and children Joel and Elizabeth whose understanding, patience and caring have made the long, hard process of college much more pleasurable and manageable.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter 1: Introduction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2: Background</td>
<td>4</td>
</tr>
<tr>
<td>Chapter 3: Automated Test Case Generation</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Test Case Generation for Spreadsheets and WYSIWYT</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1 Basic Methodology</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2 Goal-oriented Test Case Generation</td>
<td>14</td>
</tr>
<tr>
<td>3.1.3 Supplying and Using Range Information</td>
<td>22</td>
</tr>
<tr>
<td>3.1.4 Evaluation</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Chaining</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Empirical Studies</td>
<td>25</td>
</tr>
<tr>
<td>3.3.1 Subjects</td>
<td>26</td>
</tr>
<tr>
<td>3.3.2 Measures</td>
<td>27</td>
</tr>
<tr>
<td>3.3.3 Experiment Methodology</td>
<td>28</td>
</tr>
<tr>
<td>3.3.4 Experiment Design</td>
<td>32</td>
</tr>
<tr>
<td>3.3.5 Threats to Validity</td>
<td>43</td>
</tr>
<tr>
<td>3.3.6 Discussion</td>
<td>44</td>
</tr>
<tr>
<td>Chapter 4: Test Reuse</td>
<td>47</td>
</tr>
<tr>
<td>4.1 Previous Test Reuse Methodology for Spreadsheets</td>
<td>49</td>
</tr>
<tr>
<td>4.1.1 Aggressive Test Selection</td>
<td>49</td>
</tr>
<tr>
<td>4.1.2 Non-Aggressive Test Selection</td>
<td>50</td>
</tr>
<tr>
<td>4.1.3 Drawbacks</td>
<td>50</td>
</tr>
<tr>
<td>4.2 Test Re-Use Methodology</td>
<td>51</td>
</tr>
<tr>
<td>4.2.1 Data Structures</td>
<td>53</td>
</tr>
<tr>
<td>4.2.2 Saving Test Case Information</td>
<td>54</td>
</tr>
<tr>
<td>4.2.3 Responding to Changes</td>
<td>60</td>
</tr>
<tr>
<td>4.2.4 Re-using Test Cases</td>
<td>64</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.3 Case Study</td>
<td>70</td>
</tr>
<tr>
<td>4.3.1 Object of Study</td>
<td>70</td>
</tr>
<tr>
<td>4.3.2 Study Design</td>
<td>71</td>
</tr>
<tr>
<td>4.3.3 Evidence Analysis</td>
<td>72</td>
</tr>
<tr>
<td>Chapter 5: Conclusion</td>
<td>75</td>
</tr>
<tr>
<td>Bibliography</td>
<td>77</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2.1</td>
<td>Forms/3 spreadsheet Budget.</td>
</tr>
<tr>
<td>2.2</td>
<td>Forms/3 spreadsheet Budget with testing information displayed after a user validation</td>
</tr>
<tr>
<td>2.3</td>
<td>Cell relation graph for GrossPay.</td>
</tr>
<tr>
<td>3.1</td>
<td>Overall algorithm for generating a test case.</td>
</tr>
<tr>
<td>3.2</td>
<td>Goal-oriented algorithm.</td>
</tr>
<tr>
<td>3.3</td>
<td>Algorithm for minimizing the branch function for a break point.</td>
</tr>
<tr>
<td>3.4</td>
<td>Response time vs. Coverage graphs.</td>
</tr>
<tr>
<td>4.1</td>
<td>Overall algorithm for collecting test information when the user validates a cell.</td>
</tr>
<tr>
<td>4.2</td>
<td>Algorithm for collecting inputs when the user validates a cell.</td>
</tr>
<tr>
<td>4.3</td>
<td>Algorithm for updating coverage when the user validates a cell.</td>
</tr>
<tr>
<td>4.4</td>
<td>Forms/3 spreadsheet GrossPay.</td>
</tr>
<tr>
<td>4.5</td>
<td>Cell relation graph for GrossPay.</td>
</tr>
<tr>
<td>4.6</td>
<td>Algorithm for selecting re-usable tests and updating testedness information following changes.</td>
</tr>
<tr>
<td>Table</td>
<td>Title</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.1</td>
<td>Branch functions for true branches of relational operators.</td>
</tr>
<tr>
<td>3.2</td>
<td>Branch functions for logical operators.</td>
</tr>
<tr>
<td>3.3</td>
<td>Data about subject spreadsheets</td>
</tr>
<tr>
<td>3.4</td>
<td>Ultimate effectiveness of techniques per spreadsheet (mean coverage)</td>
</tr>
<tr>
<td>3.5</td>
<td>Form level coverage range</td>
</tr>
<tr>
<td>3.6</td>
<td>Median response times per Spreadsheet</td>
</tr>
<tr>
<td>3.7</td>
<td>Median response times on failure for chaining techniques</td>
</tr>
<tr>
<td>4.1</td>
<td>Data structures supporting test re-use</td>
</tr>
<tr>
<td>4.2</td>
<td>Actions per type of modification</td>
</tr>
<tr>
<td>4.3</td>
<td>Test re-use data gathered in case study on MBTI</td>
</tr>
</tbody>
</table>
HELPING END USERS CREATE AND MANAGE TEST CASES IN THE WYSIWYT METHODOLOGY

CHAPTER 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 [31], for scientific visualization [7] for providing steerable simulation environments for scientists [5], and for specifying full-featured GUIs [20, 30].

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. In essence, providing these spreadsheet formulas is an example of first-order functional programming.

It is important that spreadsheets function correctly, but research shows that they often contain faults. A survey of the literature [24] reports, for example,
that 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 60.8% of those spreadsheets; in four experiments in which participants inspected spreadsheets for faults, an average of 55.8% of faults were missed. Research has also shown that spreadsheet users tend to have unwarranted confidence in the correctness of their spreadsheets [3, 33].

In spite of such evidence, until recently, little work had been done to help end users assess the correctness of their spreadsheets. Thus, researchers have been developing a testing methodology for spreadsheets termed the “What You See Is What You Test” (WYSIWYT) methodology [25, 27, 28]. 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 [17, 29].

To test a spreadsheet the WYSIWYT methodology requires that the user find values for input cells that execute new “situations” in the spreadsheets. This can involve considerable effort in the presence of complicated formulas or in cases where a user is testing a spreadsheet that he or she did not create. Therefore, we developed the “Help-Me-Test” system, that when invoked automatically determines a set of inputs that will execute an untested situation in the spreadsheet.

Spreadsheet environments are highly interactive. This encourages a highly incremental form of development of spreadsheets where a user will iteratively create, test and modify portions of code in rapid succession. During this process a user could create test cases that would be helpful and useful to reuse.
Previously this meant that the user had to actively remember which test cases they found useful and then determine which of those would be appropriate for a given modification. To remove the need for remembering previous test cases, we have developed a strategy for the system to remember previously used test cases and to select test cases to help the user re-test the spreadsheet following modifications.

Our test generation and reuse methodologies build on initial work by Cao [6] and Jin [15]; however, this work provides new and improved algorithms, integrates the approaches more tightly and offers additional empirical results.

Chapter 2 provides necessary background about the WYSIWYT methodology. Chapter 3 briefly describes “Help-Me-Test” and automated test case generation as developed by Cao, and then describes our new generation algorithm and an empirical study examining the effectiveness and efficiency of this new algorithm. Chapter 4 briefly describes the regression testing methodology developed by Jin, and then describes our new methodology that helps alleviate some of the problems with Jin’s methodology. Chapter 5 is the conclusion.
CHAPTER 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, 7, 18, 30]) based on this paradigm.

In this paper, we present examples of spreadsheets in the research language Forms/3 [4]. Figure 2.1 shows an 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 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.

In the "What You See Is What You Test" (WYSIWYI), methodology for testing spreadsheets [25, 28, 29], 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...
FIGURE 2.1: Forms/3 spreadsheet Budget.

The following example illustrates the process. Suppose the user constructs the Budget spreadsheet by entering cells and formulas, reaching the state shown in Figure 2.1. 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.)

1 Other criteria are discussed in [27].
<table>
<thead>
<tr>
<th>Item</th>
<th>Units</th>
<th>Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pens</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paper Clips</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 2.2, 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 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 (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 ClipTotal 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 TotalCost. 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 the WYSIWYT 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]; this criterion is called the du-adequacy criterion. This criterion is defined in [25]; here, we summarize that presentation.

The du adequacy criterion is defined through an abstract model of spreadsheets called a cell relation graph (CRG). Figure 2.3 shows the CRG for spread-
sheet Budget. A CRG consists of a set of cell formula graphs (enclosed in dashed 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 [13, 32].
CHAPTER 3
AUTOMATED TEST CASE GENERATION

Prior to the creation of the "Help-Me-Test" methodology, the WYSIWYT methodology 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.

To address this problem, Cao implemented two algorithms, random and goal-oriented, for automatically generating test cases [6]. She also performed preliminary experiments to evaluate the effectiveness of the algorithms. While able to cover many of du-associations in the subject spreadsheets, her techniques' results were disappointing in several of the cases. Thus, guided by work by Ferguson and Korel [10], we have extended Cao's goal-oriented technique in hopes of achieving better performance on these spreadsheets.

In the next section we describe the overall methodology and the random and goal-oriented techniques for generation. We also summarize the results from Cao's initial experiment. Following that, we describe the "chaining" extension to the goal-oriented techniques, and we conclude this section with the results of a study comparing the new algorithm with random generation.
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 = BackupInputCells(InputCells)
4. if RandomGen(UsefulDUs, InputCells) then
5.   UpdateDisplay()
6. else
7.   RestoreConfig(InputCells, InputValues)
8. end if

FIGURE 3.1: Overall algorithm for generating a test case.

3.1 Test Case Generation for Spreadsheets and WYSIWYT

To present the methodology underlying both Cao's and our work, we begin by describing the user actions and system responses that comprise a basic version of that methodology. Sections 3.1.2 and 3.1.3 then present refinements.

3.1.1 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. Figure 3.1 provides an overview of the process the system follows to generate a test case.

The system's first task (line 1) 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, \textit{UsefulDUs} is the set of all unvalidated du-associations in the spreadsheet. If the user has selected one or more cells in the spreadsheet, \textit{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, \textit{UsefulDUs} includes each of the unvalidated du-associations associated with each such arrow.

The second task of the system (line 2) is to determine the set of input cells, \textit{InputCells}, that can potentially cause du-associations in \textit{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 [25]), it is available in data structures kept by the engine, and is retrieved from there.

Given \textit{UsefulDUs} and \textit{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 invokes a test case generation technique (line 4).

There are many test generation techniques that could be utilized; the simplest of these is to randomly generate input values. Cao implemented this technique in her prototype; we refer to it as \textit{Random}. (We use \textit{Random} here to illustrate our overall methodology; later sections describe additional techniques.)

\textit{Random} randomly assigns values to the cells in \textit{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 \textit{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, 8].

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). 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 re-
quested help testing cell BudgetOK? in Figure 2.2, the system would manipulate the values in cells PenUQ, PenUP, etc., and would present only BudgetOK? 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 (line 7). 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 (e.g. if they dislike 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.

3.1.2 Goal-oriented Test Case Generation

Random is easy to implement and provided 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). Initially, therefore, Cao adapted Korel's "Goal-Oriented Approach" [6, 16] for this purpose; we call this adaptation Goal-Oriented.

---

1 Personal communication, Jeff Offutt.
3.1.2.1 Overall Algorithm

Like Random, Goal-oriented is invoked, in our methodology, to find a set of inputs that exercise one or more du-associations in UsefulDUs. In terms of Figure 3.1, simply replace the call to RandomGen in line 5 with a call to GoalGen. Unlike Random, Goal-oriented 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, Goal-oriented terminates, and the visual devices described above for indicating relevant validatable output cells are activated. If Goal-oriented 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. Figure 3.2 provides an overview of the algorithm for Goal-oriented.

We now describe the process by which, in considering a du-association \((d,u)\), Goal-oriented 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.\(^2\) For example, to exercise du-association \((29, (23,T))\) in Budget (see the CRG in Figure 2.3 and its associated spreadsheet in Figure 2.2), input values must cause node 29 in the formula graph for UnitsError to be reached, and they must also cause the true branch of node 23 in the formula graph for cell TotalCost to be reached.

\(^2\) 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 [25].
algorithm GoalGen(UsefulDUs)
input UsefulDUs : a list of du-associations to try to cover
returns success or failure
1. for each \((d,u) \in \text{UsefulDUs}\) do
2. \(\text{finished} = \text{false}\)
3. while not \(\text{finished}\) do
4. \(b = \text{FindBreakPoint}(d,u)\)
5. \(\text{MinimizeBreakPoint}(b)\)
6. if not \(\text{Satisfied}(b)\) then
7. \(\text{finished} = \text{true}\)
8. else if \(\text{Exercised}(d,u)\) then
9. \(\text{return success}\)
10. end if
11. end while
12. end for
13. \(\text{return failure}\)

FIGURE 3.2: Goal-oriented algorithm.

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, the correct edge out of any predicate nodes lying on the direct path from \(e\) to \(d\) (or \(u\)), and \(d\) (or \(u\)). For example, the constraint path for definition 29 and use (23, T) in the CRG for Budget are \((27,(28,T),29)\) and \((22,(23,T))\), 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 \((29,(23,T))\) in Budget is \((27,(28,T),29,22,(23,T))\).

When considering du-association \((d,u)\), Goal-oriented 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 – otherwise 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. Goal-oriented's task is to alter this situation, by finding inputs that cause all nodes on the constraint path to be reached.

To do this, Goal-oriented 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 WYSIWYG subsystem. In Figure 3.2, calculating the constraint path and finding the break point occurs in the function call to FindBreakPoint in line 4. In the example we have been considering, the relevant concatenated execution trace, assuming the spreadsheet’s input cells have values as shown in Figure 2.2, is \((27,(28,F),30,31,22,(23,F),25,26)\). Goal-oriented 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 our example, the break point is \((28,F)\).

Given a break point \(b\), Goal-oriented’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 3.1.2. If the search procedure fails, we have failed on our attempt to cover this du-association, and we move on to the next du-association in UsefulDUs (line 6-7). If the break point is satisfied, then two outcomes are possible. (1) Du-association \((d,u)\) is now
exercised. In this case, the technique has succeeded and terminates (lines 8-9). (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 been satisfied (i.e., another break point exists), and (d,u) has not yet been executed. In this case, Goal-oriented repeats the above process, finding the next break point and initiating a new search, to try to make further progress.

In the example we have been considering, the only outcomes possible are that the search fails, or it succeeds causing du-association (29, (23, T)) to be exercised. If, however, cell UnitsError had contained another predicate node p in between nodes 28 and 29, such that node 29 is reached only if p evaluates to "false", then it could happen that the inputs found to cause 28 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).

3.1.2.2 The Search Procedure

The search procedure used by Goal-oriented 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 3.3.

A variation on this algorithm lets Goal-oriented 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, Goal-oriented 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 Goal-oriented 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.
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\textsuperscript{4} defined branch functions for relational operators in spreadsheets, similar to those presented in [10], as shown in Table 3.1. 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 3.2. The purpose of these functions is to allow other branch functions to be combined in a meaningful way.

After calculating the branch function for a break point, the search procedure seeks a set of inputs that satisfy that branch function \textit{without violating a constraint, in the constraint path, that is already satisfied}. This search involves a constrained linear search over inputs in $\text{InputCells}$, in which, following the procedure used by Ferguson and Korel [10], a sequence of “exploratory” and “pattern” moves are applied over time. (For efficiency, the search considers

\textsuperscript{4} Cao defined an initial set of branch functions, later we enhanced the branch functions to better support complex predicates.
<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 3.1:** Branch functions for true branches of relational operators.

<table>
<thead>
<tr>
<th>Logical Operator</th>
<th>Branch Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l ) and ( r )</td>
<td>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)) )</td>
</tr>
<tr>
<td></td>
<td>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)) )</td>
</tr>
<tr>
<td>( l ) or ( r )</td>
<td>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)) )</td>
</tr>
<tr>
<td></td>
<td>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)) )</td>
</tr>
<tr>
<td>( \neg e )</td>
<td>True branch: ( f(e, false) )</td>
</tr>
<tr>
<td></td>
<td>False branch: ( f(e, true) )</td>
</tr>
</tbody>
</table>

**TABLE 3.2:** Branch functions for logical operators.

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
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
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
20. /* Pattern Moves */
21. while delta ≠ 0 do
22.     finished = false
23.     v = CurrentValue(f)
24.     if v > 0 then
25.         return
26.     end if
27.     i = i + delta
28.     if CurrentValue(f) > v then
29.         delta = delta × 2
30.     else
31.         i = i - delta
32.         if |delta| = 1 then
33.             delta = 0
34.         else
35.             delta = delta ÷ 2
36.         end if
37.     end if
38. end while
39. end for
40. end while

FIGURE 3.3: Algorithm for minimizing the branch function for a break point.

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

In the example that we have been using, the branch function for the break point (28, F) is “if PenUO -0 ≤ 0 and ClipUO -0 ≤ 0 then 0 - PenU 0 +0 - ClipUO else 1.” The input cells that could affect this branch function are ClipUO and PenUO. Assume that the search procedure first considers the cell PenUO, and that the input cells have initial values as shown in Figure 2.2. In this case, the branch function evaluates to -120 (line 7). First the search procedure performs exploratory moves to try to determine in which direction the input should be changed (lines 9-19). Assume ClipUO 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 starts performing pattern moves in the negative direction on PenUO (lines 20-37). At each step it doubles the size of the step, until the value of the branch function no longer improves or some other constraint is satisfied. So the next step would be to change ClipUO to 17 (a step of -2). Then the branch function evaluates 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, node 28 is taking the true branch, and (29, (23,T)) is being exercised.

3.1.3 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.

Cao believed that availability and use of range information might affect the efficiency and effectiveness of test case generation techniques. Thus, her implementations of both the random and goal-oriented techniques supported the use of explicit ranges.

3.1.4 Evaluation

For [6], Cao performed a small study comparing Random and Goal-oriented. In her study, Goal-oriented achieved over 90% coverage on 7 of 10 spreadsheets without the use of explicit range information, and on 9 of 10 when explicit range information was used. However, on two spreadsheets, Grades and RandomJury, Goal-oriented performed worse than Random both with and without ranges and performed worse with explicit ranges on 6 of the other spreadsheets.

Due to these somewhat disappointing results we decided to investigate either enhancing the goal oriented methodology, or developing a new methodology.
3.2 Chaining

In [10], Ferguson and Korel extended Korel’s goal-oriented algorithm to create a new methodology for test case generation called the chaining algorithm. This methodology is similar to goal-oriented, except for its behavior when it is unable to cover a particular element. In an effort to improve on the results we observed for goal-oriented, we applied a similar extension in Forms/3.

To understand this approach, recall that in Figure 3.2, after the call to MinimizeBreakPoint in line 5, if Goal-oriented failed to satisfy that break point, it quit trying to generate a test case for that du-association. It is at this point that Chaining differs from Goal-oriented.

When MinimizeBreakPoint fails to satisfy a break point, \( b \), Goal-oriented 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 break point \( b \), Chaining collects a set \( \text{ChainDUs} \) 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 \( b \), i.e. the branch we wish to take, and (2) \( d' \) is not currently exercised. These du-associations, if exercised, necessarily enable the desired branch to be taken. Chaining iterates through du-associations in \( \text{ChainDUs} \), applying (recursively) the same process described for use on \( (d,u) \) to each.\(^5\)

\(^5\) 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.
Suppose that instead of trying to cover the du-association, \((29, (23, T))\), we had tried to cover \((24, (33, T))\). In this case, the break point would have been \((23, F)\), which would have the branch function “if \(\text{UnitsError} - 1 = 0\) then 1 else \(-|\text{UnitsError} - 1|\).” In this case, our search procedure would fail to solve this branch function (\(\text{UnitsError}\) will be a constant 0 until \(\text{PenUO}\) or \(\text{ClipUO}\) are less than 0). Therefore Chaining would “chain back” on the break point \((23, F)\), generating the set \{\((29, (23, T))\)\}, and recursively trying to cover this du-association (which we showed could be covered using Goal-oriented).

3.3 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 a sufficient number of feasible du-associations, and whether it can do so sufficiently efficiently. Also, we must discover, in cases in which the methodology fails, if it can report failure sufficiently quickly. If the answers to these questions are negative, there is no reason to pursue studies involving human subjects.

Therefore, in our initial empirical studies, we focus on these fundamental questions:

\textbf{RQ1:} Can our test case generation methodology generate test cases that execute a large proportion of the feasible du-associations of interest?
RQ2: Can our methodology generate test cases within a reasonable amount of time?

RQ3: Can our methodology respond on failure within a reasonable amount of time.

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. However for the purposes of this study we focused only on generation at the whole spreadsheet level.

3.3.1 Subjects

We used ten spreadsheets as subjects (see Table 3.3). 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 3.3 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


TABLE 3.3: Data about subject spreadsheets

<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>50</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>

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.

### 3.3.2 Measures

To investigate our research questions we use two measures: \textit{effectiveness} and \textit{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. We gather this metric incrementally over the course of repeated attempts to generate test cases for a spreadsheet, automatically recording the new cumulative coverage reached whenever a new test case is generated, and recording the final level of coverage achieved. 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. This represents the period of time a user might have to wait for a successful response, and in interactive spreadsheet systems, is crucial. We expect, however, that this amount of time will differ across du-associations, and that as the coverage of the feasible du-associations in a spreadsheet nears 100%, the time required to generate a new useful test case will increase. Thus, we gather this metric, too, over the course of repeated attempts to generate test cases for a spreadsheet until all du-associations have been exercised (or a time limit has been reached).

3.3.3 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 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 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.
3.3.3.1 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.

3.3.3.2 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 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.

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, this condition will never 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 form and cell level 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. At the DU level, we decided to use 450 seconds for all experiments, recognizing that generation should occur well before this mark.\(^6\)

\[3.3.3.3\] 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.

\(^6\) 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.
3.3.3.4 Range information

Our experiments also 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 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.)

3.3.3.5 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.

3.3.4 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.
3.3.4.1 Research Question 1

The first question to address is whether our test case generation methodology can generate test cases that execute a large proportion of the feasible du-associations of interest.

Essentially, RQ1 asks how effective our test case generation is. Here 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 at various experiment levels – we refer to this as their ultimate effectiveness. Also, we look at how consistent coverage levels attained are across all runs.\footnote{The data used to answer this research question comes from an earlier experiment with several Forms/3 features disabled. This allowed us to get higher coverage more quickly, but it did not provide accurate timings for the purpose of answering research questions 2 and 3, for which different runs were used. Further details about this experiment are available in [11].}

Ultimate Effectiveness  Ultimate effectiveness is simply a percentage measure of how many feasible du-associations in a spreadsheet can be covered by our various methods. Table 3.4 lists, for each of the subject spreadsheets, the ultimate effectiveness of Random and Chaining with and without range information, averaged across 35 runs.

As the table shows, Chaining without range information achieved over 99% 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 3.4 indicates that there was little difference in ultimate ef-
TABLE 3.4: Ultimate effectiveness of techniques per spreadsheet (mean coverage).

<table>
<thead>
<tr>
<th>Technique</th>
<th>Random</th>
<th>RandomR</th>
<th>Chain</th>
<th>ChainR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>96.57%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Digits</td>
<td>59.44%</td>
<td>97.89%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>FitMachine</td>
<td>50.50%</td>
<td>50.50%</td>
<td>97.93%</td>
<td>97.90%</td>
</tr>
<tr>
<td>Grades</td>
<td>67.10%</td>
<td>99.82%</td>
<td>99.71%</td>
<td>99.89%</td>
</tr>
<tr>
<td>MBTI</td>
<td>25.64%</td>
<td>100.00%</td>
<td>99.87%</td>
<td>99.64%</td>
</tr>
<tr>
<td>MicroGen</td>
<td>71.43%</td>
<td>99.18%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>NetPay</td>
<td>40.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>NewClock</td>
<td>57.14%</td>
<td>100.00%</td>
<td>99.01%</td>
<td>99.36%</td>
</tr>
<tr>
<td>RandomJury</td>
<td>78.78%</td>
<td>83.23%</td>
<td>94.29%</td>
<td>92.69%</td>
</tr>
<tr>
<td>Solution</td>
<td>57.69%</td>
<td>78.79%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

Effectiveness between Chaining with and without range information. In fact, on FitMachine and RandomJury, the two cases in which there was the greatest potential for improvement, addition of range information actually decreased (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 (α < .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

\(^8\)Statistical data was obtained using StatView 5.0.
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.)

Coverage Range The second view of interest in considering effectiveness is the range in coverage levels reached for each method on each spreadsheet. If the ranges are very small, and the coverage reached is fairly low, then we might wish to explore what attributes of the spreadsheet prevent coverage from exceeding a
TABLE 3.5: Form level coverage range.

certain threshold. Also, if coverage levels reached are high and fairly consistent, we might use this to aid in discovering infeasible du-associations.

Table 3.5 shows the coverage ranges reached by each technique on each spreadsheet. As the table shows, there are very few instances of large ranges of coverage reached. In fact, on all but five spreadsheets, there are no instances of coverage range greater than 5%.

We had expected to see large coverage ranges across most of Random without ranges and to a lesser extent, Random with ranges. However, Random without ranges exhibited a large range of coverage on only one spreadsheet (Digits). All other spreadsheets have range of 1.1% or less. Random with ranges, while more consistent in coverage levels than we expected, did have equal or larger coverage ranges compared to Random without ranges on all spreadsheets save Digits.
In considering Chaining with range information, we also expected range information to noticeably lower variance in coverage. This was true on all spreadsheets save FitMachine, which has a coverage range of .5% more with range than without.

We also expected that Chaining would be far more consistent than Random. This, however, was not the case in six (Grades, NetPay, FitMachine, MBTI, NewClock, RandomJury) spreadsheets both with and without range information. However, only Grades and RandomJury had over a 5% difference in coverage range on Chaining without range versus Random without range. Only RandomJury had over a 5% difference in coverage range on Chaining with range versus Random with range.

3.3.4.2 Research Question 2

The next research question that we address is whether our methodology can generate test cases within a reasonable amount of time. Inherently, this question is in regard to successful test generation runs. Thus, comparing Random with Chaining in this context is perfectly viable because we do not have to take into account the artificial method in which Random runs are stopped when unsuccessful.

Basically, RQ2 asks how long a user should expect to wait for an answer after clicking HMT in the case in which HMT is successful. First, we present median response times in order to get a feel for what is to be expected upon employing HMT.

**Successful Response - Median.** Table 3.6 presents the median response times for each method on each spreadsheet. Random with and without ranges was
very fast (less than 10 seconds) on six of the ten spreadsheets. Chaining with and without ranges achieved 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 were less than one minute, except on three of the four techniques when applied to RandomJury, where the median response time ranged from just over 80 seconds to a little over 3 minutes.

We expected to find that range information would help response times in both Chaining and Random. However, this was not generally the case. For Chaining there were only small differences with and without ranges, usually less than 1 second. For Random there were some larger differences, however
most of them favored the absence of ranges. This was probably due to the low levels of coverage that were reached by Random without ranges.

One limitation of the median response times is that it does not take into account differences in numbers of test cases generated, or coverage reached. This requires another view of the response time data, and towards that end we consider coverage graphs.

**Successful Response - Coverage Graphs.** Another technique we used to look at response times is by following response time trends as coverage increases. This helps to show places in which higher response times are acceptable. For example, if one technique takes much longer time as coverage levels grow, and another technique has small response times while not reaching the same coverage levels, larger response times may be acceptable. In order to look at this we used the graphs in Figure 3.4. These graphs were created by plotting the mean response time of all of the generated test cases that reached a level of coverage in each 10% range of coverage for each method.

A prominent point of interest in the plots is how Chaining with and without range informations seem to shadow each other throughout all graphs save two (Digits and Solution). The same is the case for Random with and without ranges, except that Random without ranges is not represented at higher coverage levels. This indicates that ranges (in general) had only slight effects on response time.

When comparing Chaining with Random, Chaining has a smaller response trend as coverage increases in six of the form-level experiments (Digits, Grades, MBTI, MicroGen, RandomJury, and Solution). The remaining four spreadsheets show varied information. Three (FitMachine, NetPay, NewClock) of the Random
FIGURE 3.4: Response time vs. Coverage graphs.
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 im-
prove to near equal or better than the Random response times. Random reaches 
similar coverage levels and maintains lower response time trends than Chaining 
in only two spreadsheets (Budget and NewClock), but only with range informa-
tion.

3.3.4.3 Research Question 3

The final research question we consider is closely related to RQ2. RQ3 asks how 
long HMT will take to respond, given that the call to HMT ended in failure, or 
no du-associations were exercised in the run. The first important item to notice 
here is that Random cannot be meaningfully represented because its stopping 
time is, by definition, hardcoded as a constant.

Failure Response - Median. Table 3.7 presents the median response times 
given that HMT has failed to exercise a du-association.

There appears to be relative consistency in this case. Seven of the spread-
sheets (Grades, MBTI, MicroGen, NetPay, NewClock, Budget, and Solution) 
show almost no discrepancy between failure response times in comparing ex-
periments using range information to those without. The final three spread-
sheet subjects (Digits, FitMachine, and RandomJury) show larger differences. 
Digits with range information takes almost twice as long to respond in failure 
than without ranges. FitMachine without ranges takes nearly 3 times as long 
as with ranges. RandomJury takes almost a factor of 100 more time without 
ranges than with ranges. In the case of RandomJury, however, this turns out to
<table>
<thead>
<tr>
<th></th>
<th>Chain</th>
<th>ChainR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>61.64</td>
<td>59.26</td>
</tr>
<tr>
<td>Digits</td>
<td>1612.55</td>
<td>2559.13</td>
</tr>
<tr>
<td>Grades</td>
<td>48.25</td>
<td>48.85</td>
</tr>
<tr>
<td>FitMachine</td>
<td>612.45</td>
<td>226.29</td>
</tr>
<tr>
<td>MBTI</td>
<td>34.42</td>
<td>46.18</td>
</tr>
<tr>
<td>MicroGen</td>
<td>11.83</td>
<td>11.75</td>
</tr>
<tr>
<td>NetPay</td>
<td>9.95</td>
<td>9.32</td>
</tr>
<tr>
<td>NewClock</td>
<td>86.44</td>
<td>89.22</td>
</tr>
<tr>
<td>RandomJury</td>
<td>1302.99</td>
<td>15.13</td>
</tr>
<tr>
<td>Solution</td>
<td>12.11</td>
<td>11.91</td>
</tr>
</tbody>
</table>

**TABLE 3.7: Median response times on failure for chaining techniques.**

be not particularly interesting, because through all of the form experiments on RandomJury, Chaining without range failed on only a single run.

Of greater interest is the relationship between failure response times, and the success response times listed in Table 3.6. In all 20 cases, median response time for a failure was higher than median response for success (sometimes nearly 100 times higher). This may indicate that setting a time limit on Chaining (possibly related to some measure of complexity of the spreadsheet) may improve failure response time without decreasing coverage.
3.3.5 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.

### 3.3.6 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. These results thus motivate further work on test case generation techniques for spreadsheets, and in the future experiments with end users to determine whether they are able to effectively use generated test cases to test spreadsheets.

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. However, Random exhibited much faster median response times, and faster response times at lower levels of coverage, which might indicate that a hybrid approach might be possible and more effective than using either approach alone.

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.

• A hybrid approach that begins with Random to increase the efficiency during the earlier portions of generation, and then utilizes Chaining to reach higher levels of effectiveness might be better than either approach separately.
CHAPTER 4
TEST REUSE

The incremental, responsive nature of the spreadsheet paradigm makes spreadsheets highly malleable, and spreadsheet users often make small changes to formulas in established spreadsheets. Such changes can render previously correct spreadsheets faulty, and suggest the need for re-testing. The WYSIWYGT methodology helps with this by indicating the areas of spreadsheets that are affected by changes; end users can then retest those affected areas.

Such retesting, however, can involve considerable effort. This is due partly to the difficulty of finding test inputs that exercise affected areas of spreadsheets, a task that automated test input generation can help with. However, automated test input generation may not produce outputs that can be easily validated (judged correct), forcing end users to repetitively call input generators until more suitable inputs are found, or even to generate inputs manually. When a spreadsheet is changed, it may be much simpler to re-use test cases that users have developed and found useful previously; for such test cases the correctness of outputs can be more easily established.

Further, research shows that spreadsheets are very often used and shared by other users, and passed on to still other users [21]. Sometimes these users must test a spreadsheet they have been given in order to verify its suitability for their purposes [21]. If they then decide to further adapt the spreadsheet to their own requirements, further re-testing is required. Asking users to test or re-test
“from scratch” a spreadsheet they did not write wastes resources, and puts the onus of testing on persons who may have the greatest difficulties finding useful test inputs. In such a scenario, saved test cases embedded in the spreadsheet by an experienced spreadsheet developer might more easily be selectively re-used by end users following their modifications.

Finally, production spreadsheets are processed by commercial spreadsheet engines that are periodically re-released to provide new functionality or operate on new platforms. New releases of these engines, however, can cause spreadsheets to function differently than previously; thus, organizations that use spreadsheets for safety-critical tasks insist on revalidating their spreadsheets on new releases of spreadsheet engines, prior to allowing their use on those new releases.¹ Such revalidation would be greatly aided by the ability to re-use existing saved test suites.

Therefore, we have been investigating ways to add support for test re-use into our spreadsheet methodology. An initial methodology for saving and reusing tests for spreadsheets was developed by Jin [15]. However we discovered that a more integrated approach might be more consistent, and thus, we have developed a new methodology with associated algorithms for supporting test case reuse.

In this chapter we briefly describe Jin’s methodology, and then present our new methodology along with an empirical study examining a metric of that methodology’s effectiveness.

¹ Personal communication, Scott Hutchinson, Amgen Corp..
4.1 Previous Test Reuse Methodology for Spreadsheets

Jin developed two slightly different techniques for saving and re-using test cases, aggressive test selection and non-aggressive test selection, which correspond to two different behaviors that were available in Forms/3 for responding to changes in formulas. The following data structures are used for both techniques.

Every time a user places a checkmark or x-mark in a decision box, as described in Chapter 2, they are performing a test on a spreadsheet. For each of these tests we need to save the relevant information. This is done through the use of several lists that are maintained for each cell and each test.

Each non-input cell in the spreadsheet has six lists associated with it: self-ValidList, self-BugList, self-RerunList, inter-ValidList, inter-BugList, and inter-RerunList. The “self” lists include all of the tests that were actually placed on that particular cell. The “inter” lists include all of the tests for which that cell provided some value. Similarly, each test includes two lists, inputCells and intermediateCells, and also records the outputCell for the associated check- or x-mark.

4.1.1 Aggressive Test Selection

After tests are recorded, no further processing is required until a modification has been made to the spreadsheet. When a cell is deleted, all associated tests in that cell’s self lists are deleted (in Forms/3 it is not possible to delete cells that are referenced by other cells, so the inter lists of any deleted cell should be empty). If a cell’s formula is modified, and any new input cells are referenced, then all of the tests in that cell’s self lists are deleted, and each cell that references the modified cell is processed as if its formula had changed. If a
cell’s formula is modified without adding any new references to input cells, then each test in the cell’s `selfValidList` and `selfBugList` is moved to the cell’s `selfRerunList`, and each cell that references the modified cell is processed as if its formula had changed.

### 4.1.2 Non-Aggressive Test Selection

Non-aggressive test selection works similarly to aggressive test selection. The primary difference is that when a cell is modified, after deleting or moving the tests in that cell’s `self` lists, instead of recursively processing the cells that statically reference the modified cell, the system instead processes the tests in the `inter` lists, either moving the tests to the appropriate rerunnable test lists or deleting them.

### 4.1.3 Drawbacks

Jin’s techniques and formula editing suffered from the fact that they were only loosely integrated with each other. The most important drawbacks were:

- the changes in testedness information presented to the user did not agree with the set of tests selected for reuse;

- the two algorithms for responding to changes in formulas both suffered from being inaccurate, aggressive indicated that retesting was required in tested areas of the spreadsheet that could not have been affected by a formula edit, and non-aggressive failed to indicate that retesting was required in portions of the spreadsheet that could have been affected by a formula edit;
• although both of Jin's algorithms for test selection select all possibly changed test cases, both of them also select some test cases that should not need to be rerun.

In order to help with these problems we developed a new test selection technique that has a higher degree of integration with the WYSIWYT system.

4.2 Test Re-Use Methodology

In designing a test re-use methodology and incorporating it into the WYSIWYT approach, there are a number of requirements we believe necessary to consider in addition to addressing the drawbacks just described.

First, to support the highly interactive spreadsheet programming environment and its reliance on immediate feedback, a test re-use methodology for spreadsheets must be considerate of users' time and attention. This means that it must be computationally efficient, and avoid unduly delaying users creating or modifying spreadsheets from making progress on the tasks they are trying to accomplish by using the spreadsheet in the first place. This also means that a methodology should not unduly control how users spend their attention [2]: attempting to force a user to perform a potentially lengthy regression testing task could be counterproductive to the user's spreadsheet efforts at that point in the spreadsheet's development. In such cases, users may abandon the use of the methodology.

It follows that a test reuse strategy for spreadsheets must be sufficiently precise in estimating affected areas of spreadsheets and making test cases available for re-use. If an end user must perform actions that produce no apparent ben-
efit, such as retesting portions of a spreadsheet that do not apparently require retesting, they may become frustrated and choose not to re-test at all.

Further, although a test re-use strategy should be precise in the sense just described, it must also be conservative enough, in its estimates, to merit user trust. For example, in responding to user modifications, a re-use strategy should not miss areas of the spreadsheet that might be affected. Such an omission might cause end users to become (at least initially) overconfident in their spreadsheet, and later (on finding their confidence misplaced) to distrust the approach.

Finally, a test re-use methodology for spreadsheets should not, in general, require its users to practice (in any formal way) “software engineering”, or have any formal knowledge of testing theory. The users of the methodology might not even think of “test cases” as “resources to be re-used”, and they should not be expected to possess any recorded specifications that could be used to determine expected test results.

Keeping the foregoing considerations in mind, we have developed algorithms supporting test re-use in spreadsheet validation that are integrated with our WYSIWYT methodology. As we shall show in the following subsections however, the foregoing considerations cause our approach to differ from those that might be used in application to imperative programs and professional programmers.

Our first algorithm operates when an end user validates a cell, collecting information about the test case associated with that validation and storing it for later use. Our second algorithm operates when an end user changes a cell formula, determining impacted portions of the spreadsheet and effects on test-edness, and judging whether existing test cases may be applicable to retesting the impacted portions. (These first two algorithms partially replace, and par-
Information kept per cell

<table>
<thead>
<tr>
<th>Cell.Name</th>
<th>identifier for Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Value</td>
<td>a value displayed in Cell</td>
</tr>
<tr>
<td>.Trace</td>
<td>definition and use nodes exercised in Cell’s most recent execution</td>
</tr>
<tr>
<td>.ReachingTests</td>
<td>set of Tests that exercise Cell</td>
</tr>
</tbody>
</table>

Information kept per test case

| Test.InputCells | set of (Cell.Name,Cell.Value) pairs serving as inputs for Test |
| .ValidatedCell  | (Cell.Name,Cell.Value) pair for the cell validated in Test |
| .ReachedCells   | set of Cells exercised by Test |
| .ValidatedDUs   | set of du-associations Test validates |

Information kept for the spreadsheet

| DUTable       | hash table of validation counts, indexed by du-associations |
| ImpactedTests | set of Tests impacted by changes, and that might be rerunnable |

TABLE 4.1: Data structures supporting test re-use.

Initially augment, previously presented WYSIWYT algorithms not supporting test re-use; portions of the overall approach not germane to an understanding of the test re-use issue that is our focus in this paper are omitted for simplicity.) The third algorithm operates when an end user requests help re-testing the spreadsheet, replaying saved test cases so that the user can use them to revalidate the spreadsheet. The sections that follow present the data structures used by these algorithms, and then present each of the algorithms in turn.

4.2.1 Data Structures

Our test reuse algorithms rely on the data structures shown in Table 4.1 to record information on test cases and testedness and support test re-use; these data structures are maintained within the spreadsheet engine, and in most cases, updated incrementally as end users create, modify, and validate their spreadsheets. The data structures contain (1) information kept for each cell, (2) information kept for each test case, and (3) information kept for the spreadsheet.
algorithm Validate(C)
input C : cell
1. T = new Test
2. T.ValidatedCell = (C.Name,C.Value)
   /* Phase 1 */
3. InitWalk
4. GatherInputs(C,T)
   /* Phase 2 */
5. InitWalk
6. ValidateCoverage(C,T)

FIGURE 4.1: Overall algorithm for collecting test information when the user validates a cell.

4.2.2 Saving Test Case Information

When an end user performs a test by validating a cell in their spreadsheet, we walk backwards through the CRG for that spreadsheet, updating information associated with the test and its coverage, and saving the test case defined by the users' action. Our algorithm, Validate (Figure 4.1), takes as input a cell C validated by an end user. Validate first creates a new Test structure to represent the test case just created by this validation, initializing its fields to empty, and setting its ValidatedCell field to (C.Name,C.Value). The algorithm then proceeds in two phases, invoking procedures GatherInputs (Figure 4.2) and ValidateCoverage (Figure 4.3) in turn. Each invocation is preceded by

2 These two phases, each performing a graph walk on the CRG, can be merged into a single graph walk with the same worst-case cost as that incurred by performing the phases separately, but with a lower constant; however, we present them as separate phases for simplicity.
algorithm GatherInputs($C, T$)
  input $C$: cell
           $T$: test
  1. MarkVisited($C$)
  2. if IsInput($C$)
     3. $T.InputCells = T.InputCells \cup \{(C.Name, C.Value)\}$
  4. else
     5. for each cell $D \in \text{DirectProducers}(C)$
     6. if not Visited($D$)
     7. GatherInputs($D, T$)
     8. endif
     9. endfor
  10. endif

FIGURE 4.2: Algorithm for collecting inputs when the user validates a cell.

InitWalk, which initializes an integer flag used to mark cells “visited” and avoid revisiting cells during the ensuing walk.\(^3\)

GatherInputs, called with validated cell $C$ and test case $T$, performs a static backwards slice on cell dependence edges in the CRG, locating (recursively) all input cells whose values could affect $C$. (We refer to cells affecting $C$ as producers of $C$. Direct producers of $C$ — maintained by the spreadsheet engine and returned

---

\(^3\) This integer flag is initialized to zero by the spreadsheet engine at startup, and incremented on each subsequent call to InitWalk; algorithms that walk the CRG use this flag to mark cells “visited” and avoid revisiting cells during a given walk. This eliminates the need to reset “visited” information on all cells prior to each walk, except in the unlikely event that the integer flag reaches its maximum value during a session using the spreadsheet engine.
algorithm ValidateCoverage($C, T$)
input $C$: cell
      $T$: test
1. MarkVisited($C$)
2. $C.$ReachingTests = $C.$ReachingTests $\cup$ $T$
3. $T.$ReachedCells = $T.$ReachedCells $\cup$ $C$
4. for each use $u$ in $C.$Trace
5. $D = $ the cell referenced in $u$
6. $d = $ the current definition of $D \in D.$Trace
7. increment $D.$UTable($d, u$)
8. $T.$ValidatedDUs = $T.$ValidatedDUs $\cup$ {$($d, u$)$}
9. if not Visited($D$)
10. ValidateCoverage($D, T$)
11. endif
12. endfor
13. UpdateDisplay($C$)

FIGURE 4.3: Algorithm for updating coverage when the user validates a cell.

by a call to a function DirectProducers in line 11 — are producers explicitly
listed in $C$’s formula.)

ValidateCoverage, called with validated cell $C$ and test case $T$, performs
a dynamic backwards slice on du-associations in the CRG starting at $C$, re-
cursively visiting each cell reached by $T$. Specifically, for each cell reached, the
procedure adds $T$ to that cell’s list of reaching tests (line 18), and adds that
cell to $T$’s list of reached cells (line 19). The procedure next uses trace informa-
tion maintained by WYSIWYT (collected and maintained by the spreadsheet

$^{4}$ Using a static rather than a dynamic slice in this phase reduces the number of test cases
that must be discarded as obsolete following a formula modification – the larger input cell
sets recorded by the static slice reduce the likelihood that changes to a cell formula will
increase the set of input cells on which that formula depends, making test cases associated
with that cell non-applicable. We discuss test obsolescence and our handling of it in Section
4.2.4.
FIGURE 4.4: Forms/3 spreadsheet GrossPay.

engine during its operations to update the display following formula changes) to locate the du-associations ending at the cell reached that were exercised by $T$. For each such du-association $(d,u)$, the procedure increments the coverage count for $(d,u)$ (line 23), and adds $(d,u)$ to the list of du-associations covered by $T$ (line 24). Finally, the procedure calls UpdateDisplay to cause the colors of the cell’s border, and of any arrows ending at that cell and currently being displayed, to be updated.

To illustrate how this algorithm works, consider the GrossPay spreadsheet shown in Figure 4.4, and its CRG shown in Figure 4.5. When the user clicks on the decision box for cell GrossPay, Validate is called for that cell. The algorithm first creates a new Test $T$, and sets $T$.ValidatedCell to (GrossPay,
FIGURE 4.5: Cell relation graph for GrossPay.

380). It then calls GatherInputs on GrossPay and T. Since GrossPay is not an input cell, GatherInputs takes the else branch from line 8, and then iterates over the set DirectProducers(GrossPay), containing cells PayRate and
TotalHours, recursively calling GatherInputs on each. PayRate is an input cell, so T.InputCells is set to \{(PayRate, 10)\}. Next, the recursive call GatherInputs(TotalHours, T) iterates over DirectProducers(TotalHours), adding (Mon, 8), (Tues, 8), (Wed, 8), (Thurs, 7), (Fri, 7) to T.InputCells.

Validate next calls ValidateCoverage with GrossPay and T. ValidateCoverage adds T to GrossPay.ReachingTests, and adds GrossPay to T.ReachedCells. It then iterates through the use nodes, (23, TotalHours, T), (24, PayRate), and (24, TotalHours), found in GrossPay.Trace. On the first use, (23, TotalHours, T), the procedure finds that the currently active definition of TotalHours is node 17, and increments the testedness counter in DUTable(17, (23, TotalHours, T)). It adds (17, (23, TotalHours, T)) to T.ValidatedDUs, and recursively calls ValidateCoverage(TotalHours, T), which updates TotalHours.ReachingTests and T.ReachedCells, and iterates through the five uses in TotalHours. When the iteration in ValidateCoverage(GrossPay, T) reaches (24, PayRate), (and subsequently (24, TotalHours)), a similar procedure is performed (except that ValidateCoverage is not recursively called on TotalHours as it was already marked “visited”).

The time complexity for Validate is determined by the cost of its calls to GatherInputs and ValidateCoverage. The cost of the initial call to GatherInputs is determined by the number of times it is recursively called and the cost of each call. Since GatherInputs marks each cell when visited, and is not recursively called on visited cells, the upper bound on the number of calls to GatherInputs is \(p\), where \(p\) is the number of producers of the validated cell. Each call to GatherInputs iterates through each of a cell’s direct producers even if it does not make a recursive call, so this places an upper bound on iterations in a call to GatherInputs at \(d\), where \(d\) is the maximum number
of direct producers of a cell in the spreadsheet. Each iteration involves a set union; however, the sets unioned are always necessarily disjoint. If these sets are implemented as lists and the union is implemented as an append, each iteration can be accomplished in constant time. Thus, GatherInputs runs in time $O(dp)$.

Similarly, the cost of an initial call to ValidateCoverage is determined by the number of recursive calls that follow and the cost of each call. Since ValidateCoverage marks each cell when visited and is not recursively called on visited cells, the upper bound on the number of calls to ValidateCoverage is $p$ where $p$ is the number of producers of the initial validated cell. Each call to ValidateCoverage with cell $C$ and test case $T$ iterates through each use that was reached in $C$ on $T$, even if it does not require a recursive call, so this places an upper bound on iterations of the loop at lines 20-28 in a single call at $u$, where $u$ is the maximum number of uses in a formula. Assuming an efficient hash table implementation and given that the sets unioned are disjoint and can be implemented as lists with the union implemented as a list append, each iteration of the loop can be accomplished in constant time. Thus, ValidateCoverage runs in time $O(up)$.

4.2.3 Responding to Changes

When a user modifies a spreadsheet, we must update all testing-related information, in the spreadsheet, that may be affected, and determine test cases that may be used to re-test affected portions of the spreadsheet. The actions required depend on the type of modification. Table 4.2 categorizes the modification types we need to consider, which depend on whether the user adds a cell, deletes a cell, or modifies an existing cell's formula, and on whether the cell in question
<table>
<thead>
<tr>
<th>Type of Change</th>
<th>Type of Cell</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Deletion</td>
<td>input</td>
<td>(1) Not possible, or no action needed. An input cell cannot be deleted if it is currently being referenced; an input cell not being referenced can be deleted, but due to other operations can have no test cases associated with it.</td>
</tr>
<tr>
<td></td>
<td>intermediate</td>
<td>(4) Not possible. A cell cannot be deleted if it is currently being referenced.</td>
</tr>
<tr>
<td></td>
<td>output</td>
<td>(7) Treat as formula change to blank, and use algorithm ProcessMod</td>
</tr>
<tr>
<td>Formula Change</td>
<td>formula</td>
<td>(2) No action needed or use algorithm ProcessMod. A formula change involving insertion of a new constant is simply a test execution; insertion of a formula that references other cells is handled by ProcessMod.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(5) Use algorithm ProcessMod.</td>
</tr>
<tr>
<td>Cell Insertion</td>
<td>insertion</td>
<td>(3) No action needed. An input cell being inserted cannot have any referencing cells prior to insertion; thus, its insertion changes no test information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(6) Cell insertion involves inserting blank cells, then updating formulas for those cells; thus, this situation reduces to entry (3) followed by entry (2).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(9) Cell insertion involves inserting blank cells, then updating formulas for those cells; thus, this situation reduces to entry (3) followed by entry (2).</td>
</tr>
</tbody>
</table>

TABLE 4.2: Actions per type of modification.

is an input cell, intermediate cell, or output cell (cell not referenced by other cells).

For each entry in this table, we determined the actions necessary to handle the type of modification; the table summarizes those actions. (These actions hold for Forms/3 spreadsheets; a similar approach could be used to classify and determine how to handle modification types in other spreadsheet environments.) As the table and the following discussion show, several types of modifications
**Algorithm** ProcessMod($C$)

**Input**  
$C$: cell

1. **foreach** test case $T \in C.ReachingTests$
2. **foreach** cell $D \in T.ReachedCells$
3. $D.ReachingTests = D.ReachingTests \setminus \{T\}$
4. $UpdateDisplaySet = UpdateDisplaySet \cup \{D\}$
5. **endfor**
6. **foreach** $(d,u) \in T.ValidatedDUs$
7. decrement $DUTable(d,u)$
8. **endfor**
9. $T.ValidatedDUs = \emptyset$
10. $T.ReachedCells = \emptyset$
11. $ImpactedTests = ImpactedTests \cup \{T\}$
12. **endfor**
13. $UpdateDisplay(UpdateDisplaySet)$

**FIGURE 4.6:** Algorithm for selecting re-usable tests and updating testedness information following changes.

do not require any handling, and those requiring handling are all processed by algorithm ProcessMod, shown in Figure 4.6. The actions and their use of the algorithm form a set of invariants which, together, ensure correct updating of testing-related information across possible change types, and ensure identification of test cases potentially useful for retesting.

ProcessMod relies on the fact that when a formula for a cell is edited, we already possess, attached to that cell, a list of all test cases that exercise that formula; this list includes all test cases that could be affected by the edit. The outer loop of the algorithm processes these test cases one at a time. For each test case $T$, the algorithm first removes $T$ from the $ReachingTests$ lists on each other cell it is associated with (line 3), and then adjusts $T$'s coverage information in $DUTable$ (lines 6-8). ProcessMod saves the names of those cells (line 4) for use
later in updating the display (line 13). The algorithm also sets ValidatedDUs and ReachedCells information for T to empty, since it cannot predict what behavior the test case will possess following modifications.

In line 11, ProcessMod lists T as impacted. Note that at this stage in our approach, such test cases are simply inserted into ImpactedTests, where they may join other test cases placed there following previous modifications. Further processing of these test cases, however, is postponed to the test re-use phase, as described in Section 4.2.4. Note further that the test case’s InputCells and ValidatedCell information are not altered by ProcessMod, even though, depending on the modification, that information may no longer be valid. This will also be explained in Section 4.2.4.

By identifying as impacted all test cases that exercise modified formulas, ProcessMod ensures selection of all test cases whose execution traces can differ as a result of the modification. Following results presented in [26], this allows the algorithm to not omit, in this stage of our methodology, test cases that could expose faults related to the user’s formula modification. However, adjusting coverage data to reflect removal of such test cases also ensures that the algorithm will err only conservatively in estimating the impact of the modification on testedness, and in displaying affected areas of the spreadsheet to the user. As stated earlier, this is an important factor in limiting user distrust.

As an example of the algorithm’s operation, suppose an end user modifies the formula for TotalHours in spreadsheet GrossPay. This modification is handled by entry (5) in Table 4.2, resulting in a call to ProcessMod(TotalHours). ProcessMod iterates over the set of test cases in TotalHours.ReachingTests. Suppose this set contains just the single test case described in Section 4.2.2. The algorithm decrements the coverage counts associated with this test case in
The time complexity for ProcessMod is determined by the number of iterations of each of its loops and the cost of each iteration. The outermost loop iterates \( t \) times where \( t \) is the number of test cases in \( C.ReachingTests \). Within this loop there are two separate nested loops. The first (lines 2-5) iterates at most \( n \) times where \( n \) is the maximum number of cells in the ReachedCells field of a test case. The second loop (lines 6-8) iterates for each du-association in the ValidatedDUs field of a test case. Each test case can validate at most \( O(nu) \) du-associations, where \( u \) is the maximum number of uses for the cells reached by a test case. An efficient hash table implementation holds the cost of each such iteration to \( O(1) \). Combining the above costs yields a total cost of \( O(tn^* \max(u, \text{cost of set operations})) \).

4.2.4 Re-using Test Cases

Our algorithms for saving test cases and for determining impacted test cases following modifications provide two classes of test cases that can be used to re-validate spreadsheets, in two different scenarios.

1. Test cases that have been inserted into ImpactedTests are related to modifications, and may be useful in revalidating du-associations associated with changes.

2. All saved test cases can be replayed, allowing a spreadsheet to be revalidated in full when ported to a new spreadsheet environment.

We now describe our handling of these two test classes and scenarios in turn.
4.2.4.1 Retesting Following Changes

Our algorithms cause test cases that might be useful in re-testing modified formulas and re-establishing coverage to be placed in ImpactedTests. Replaying such test cases involves applying the saved input values to appropriate input cells, and allowing the user to re-validate output cells. There are several aspects to this process that need to be considered.

Test case replay strategy. The first aspect involves choosing a test case replay strategy and designing an interface that implements that strategy.

One possible strategy, based on typical techniques used in the imperative programming paradigm [22], is to require the user to replay and revalidate all impacted test cases after each modification (or possibly after indicating that a set of related modifications has been completed). This strategy offers the greatest level of assurance and conservatism, by requiring the user to rerun all test cases that might be affected by their modifications, and might thus conceivably expose faults related to those modifications.

Reflecting on the considerations with which we began this section (Section 4.2), however, this strategy has drawbacks. Re-running all test cases identified by ProcessMod might require more time than a user is willing to spend. Further, it is possible that, due to changes in the spreadsheet or the order in which test cases are rerun, some test cases identified as “impacted” will not lead to increases in coverage, and this violates the restriction on forcing users to perform actions that produce no apparent benefit.

A second strategy involves requiring the user to revalidate all impacted test cases that could increase coverage. These test cases can be determined automatically by the system by having it save current input values, apply the
test inputs, activate the evaluation engine, and determine whether uncovered
du-associations are covered, and only then update the display and allow the
user to proceed with validation. (Considering existing imperative language ap-
proaches, this strategy is most similar to those presented in [14, 23].) While this
approach solves the problem of perceived wasted effort, it violates the principle
of allowing users to control how they spend their own attention [2].

The key drawback behind these two strategies is that the system does not
really have the power to “require” testing-related actions, because an end user
could simply turn off the feature or turn to a different spreadsheet environment
that does not have such a requirement. The action must be deemed worthwhile
to keep the user from doing this, and requirements that the user consider a
large number of test case applications, or consider test case applications that
are unproductive, is not likely to be deemed worthwhile.

A third strategy involving making only potentially useful test cases available
without forcing any particular user action can be achieved by integrating test
re-use with automated test case generation. This approach is triggered when
the user requests help generating test inputs, which they might do following a
modification to restore testedness in affected areas of the spreadsheet. Given
such a request, rather than immediately attempting to generate new inputs,
the test case generation functionality first tries to find an existing rerunnable
test case that will increase coverage of the portion of the spreadsheet indicated
by the user. If a useful test case is found, the test generation facility applies
that test case’s inputs, displays the resulting changes, and the user can then
re-validate cells under that input set.
This third strategy avoids the drawbacks of the first two, and also avoids requiring a user to have any concept of a test case as an existing, re-usable resource, instead, integrating test re-use with the “Help-Me-Test” facility. We therefore implemented this third strategy in our prototype.

Obsolete test cases. The second aspect of test case re-use to consider involves test cases that can no longer be applied as first recorded, due to changes in the spreadsheet. If the output cell associated with a test case is deleted, that test case can no longer be applied. If formula changes cause a cell to depend on new input cells, test cases formerly associated with that cell no longer come with all necessary inputs. In the literature on regression testing of imperative programs, such test cases are known as obsolete [19]. Such test cases could be detected and eliminated by ProcessMod when processing a modification; however, this adds to the expense of that algorithm, and since subsequent modifications may render a test case previously judged obsolete valid again, we defer the handling of these test cases to the test replay stage.

In our approach, when a user selects Help-Me-Test, as the test case generation facility iterates over ImpactedTests seeking useful test cases, it begins by determining whether each test case $T$ it encounters is obsolete. To do this, the test case generation facility first determines whether the validated cell $V$ associated with $T$ still exists in the spreadsheet. If not, then $T$ is obsolete and is discarded. If $V$ exists, the generator determines the set of input cells that could affect $V$, using an algorithm similar to GatherInputs. If this set includes input cells not found in set $V.InputCells$ associated with the test case originally, we do not know what values to use for those cells and cannot re-apply the test case.
transparently, so we consider it obsolete.\textsuperscript{5} Obsolete test cases can be discarded from the data structures, or retained in case further modifications render them non-obsolete; to avoid a potentially excessive build-up of such test cases we take the first course of action.\textsuperscript{6}

**Output validation strategy.** When test cases are saved, their validated output cell values are also saved; these values could be re-used in re-testing the spreadsheet after modifications. Such an approach is typical in testing imperative programs [22].

It is tempting to consider automatically checking test case outputs against saved outputs when they are re-run, and when saved outputs match new outputs, consider test cases to have automatically passed. This strategy, however, is inappropriate. An end user might modify a spreadsheet in order to change its functionality, such that previous inputs are expected to produce different outputs than previously. Thus, for some test cases, the fact that saved outputs match new outputs does not signal correctness, but rather, incorrectness. In general, in the absence of specifications, a test replay strategy cannot automatically infer that matching outputs signal either correctness or incorrectness.

\textsuperscript{5} Alternative approaches are possible. If the input cell set has increased, we can apply those inputs that we know, leaving other inputs at their current values; and if the validated cell is now absent, the test case’s input set might nevertheless still be applied. In one other scenario – the scenario in which the set of input cells now associated with a test case is a subset of those previously associated with it, we do choose such an alternative, and apply previous inputs to those input cells that still exist. Future work will examine the cost-benefits of these alternatives.

\textsuperscript{6} As foreshadowed in Section 4.2.2, it is to reduce the incidence of obsolete test cases that the first phase of Validate uses a static slice rather than a dynamic slice to determine the input cell set associated with a test case.
An alternative strategy, when replaying previous test cases, is to display the previous test case output alongside the new test case output, to help the user validate the new output. It is not clear, however, whether such an approach would actually assist the user, or whether users would be inclined to make the potentially erroneous assumption that old values should match new, and that matching indicates correctness — an assumption that, as just discussed, may be erroneous.

For these reasons, we do not yet use previous test case outputs during test replay; we intend to first explore the ramifications of possible approaches empirically with end users before selecting an output re-use strategy.

4.2.4.2 Re-applying Tests

By saving all test case information with a spreadsheet when we save it to disk, and re-loading it when that spreadsheet is re-loaded, we also support the complete re-execution of saved test suites. End users wishing to re-validate a spreadsheet for a new release of an evaluation engine can take advantage of this. This test case re-use scenario differs from the previous scenario in that it does imply that users be aware of test cases as objects, and consider saved test cases a resource; however, we expect that in certain contexts, for certain users, this will not be inappropriate.

Also, in this context, re-use of output values may make more sense, and a utility that iterates through all saved test cases may be palatable to end users. Our future studies will consider this possibility.
4.3 Case Study

A primary goal of our test case re-use methodology is to help users re-test their spreadsheets following modifications. Determining whether the methodology achieves this goal will require empirical studies of users; however, before undertaking such studies we must first address a more fundamental question: namely, whether the methodology can in fact usefully select test cases for re-use. If the answer to this question is negative, there is no reason to pursue more expensive studies involving human subjects.

To assemble some initial data toward this question, we prototyped our test re-use methodology in Forms/3, in conjunction with our WYSIWYAT and automated test case generation methodologies. We used this prototype to perform a case study of the application of our methodology to a large spreadsheet undergoing correction for errors.

4.3.1 Object of Study

As an object of study we selected a large spreadsheet, MBTI, that had been used in earlier empirical studies of automatic test case generation. Section 3.3.1 includes details on this spreadsheet.

We asked an experienced spreadsheet user unfamiliar with our test case reuse methodology and the anticipated case study to generate 14 different faulty versions of MBTI by inserting minor faults in the spreadsheet.

We used our automated test case generation system to generate a test suite for each of the faulty versions through repeated applications of the generation algorithm, continuing until several applications failed to find new useful inputs. The resulting test suites, although not 100% adequate in covering all feasi-
ble du-associations in the erroneous spreadsheets, were nearly so, with coverage ranging from 93.11% to 98.85% of the du-associations in the spreadsheets. (The differences in testedness across the spreadsheets can be attributed primarily to differences in feasible du-associations among the faulty versions, and to differences in the ability of the automated test case generation system to cover the du-associations that differed across those faulty versions.)

4.3.2 Study Design

To investigate our research question we collected data on several measures related to our test re-use methodology and the potential usefulness of the test cases our algorithms identify both following modifications, and at the time test cases are re-run. These measures include:

- the number and percentage of test cases identified by our test re-use strategy of Section 4.2.4 as rerunnable following a modification to a spreadsheet, and an end user's request to re-validate the spreadsheet;
- the amount of du-coverage that could be regained on the spreadsheet by rerunning all test cases identified as rerunnable.

To evaluate our test reuse methodology relative to these measures, we wished to simulate a user performing a correction of a faulty formula in MBTI, and then re-running all re-runnable test cases that can add coverage. To do this, for each of our faulty versions of MBTI, we loaded that version, with all of its saved test cases, into the Forms/3 environment. We then corrected the fault in the spreadsheet, causing ProcessMod to be invoked and identify impacted test cases. Finally, we ran a script that automatically performed a "Help-Me-Test" operation repeatedly, invoking the test case generation technique until all
<table>
<thead>
<tr>
<th>Version</th>
<th>Test Cases</th>
<th>Initial Coverage</th>
<th>Coverage After Correction</th>
<th>Impacted Test Cases</th>
<th>Rerunnable Test Cases</th>
<th>Obsolete Test Cases</th>
<th>Final Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>464</td>
<td>98.21</td>
<td>88.64</td>
<td>51</td>
<td>42</td>
<td>9.05</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>474</td>
<td>96.30</td>
<td>87.88</td>
<td>46</td>
<td>46</td>
<td>9.71</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>479</td>
<td>98.09</td>
<td>89.29</td>
<td>57</td>
<td>57</td>
<td>11.90</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>488</td>
<td>97.83</td>
<td>95.41</td>
<td>21</td>
<td>21</td>
<td>4.30</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>428</td>
<td>98.41</td>
<td>90.69</td>
<td>21</td>
<td>16</td>
<td>3.74</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>445</td>
<td>97.83</td>
<td>89.29</td>
<td>41</td>
<td>0</td>
<td>0.00</td>
<td>41</td>
</tr>
<tr>
<td>7</td>
<td>493</td>
<td>98.60</td>
<td>89.16</td>
<td>53</td>
<td>0</td>
<td>0.00</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td>433</td>
<td>97.99</td>
<td>87.25</td>
<td>39</td>
<td>0</td>
<td>0.00</td>
<td>39</td>
</tr>
<tr>
<td>9</td>
<td>494</td>
<td>97.45</td>
<td>89.92</td>
<td>45</td>
<td>45</td>
<td>9.11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>447</td>
<td>97.58</td>
<td>85.46</td>
<td>54</td>
<td>54</td>
<td>12.08</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>472</td>
<td>98.85</td>
<td>86.48</td>
<td>65</td>
<td>58</td>
<td>12.29</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>448</td>
<td>98.45</td>
<td>89.67</td>
<td>44</td>
<td>39</td>
<td>8.71</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>465</td>
<td>98.60</td>
<td>87.50</td>
<td>60</td>
<td>37</td>
<td>7.96</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>402</td>
<td>93.11</td>
<td>86.35</td>
<td>36</td>
<td>31</td>
<td>7.71</td>
<td>0</td>
</tr>
</tbody>
</table>

**TABLE 4.3**: Test re-use data gathered in case study on MBTI.

test cases had been considered, and judged obsolete or, if able to add coverage, re-applied. Following each application of a re-used test case, we automatically validated the output cell associated with that test case. We repeated this process on each of the fourteen faulty versions of MBTI, in turn.

### 4.3.3 Evidence Analysis

Our case study involves only a single spreadsheet, and a set of modifications made to correct relatively small faults, and thus, cannot generalize to the range of situations in which we would like to apply our approach. Keeping this limitation in mind, however, the data gathered do let us draw some observations about the feasibility of the approach and its possible benefits.
Table 4.3 presents the data collected in our study. For each faulty version of METI, the table lists the total number of associated test cases (column 2), the initial testedness these test cases achieved on the faulty version prior to correction of the fault (column 3), and the spreadsheet's reduced level of testedness following the fault correction (column 4). Column 5 shows the number of impacted test cases identified by ProcessMod. Columns 6 and 7 show the number and percentage of test cases identified as rerunnable by the methodology, and column 8 shows the number of test cases identified as obsolete. Column 9 displays the final testedness achieved on the spreadsheet after re-running the rerunnable test cases.

First we consider the number of rerunnable test cases. There are two questions to consider here: are useful rerunnable test cases available, and is the number of such test cases a reduction over the entire test suite? For three of the 14 modified versions of MBTI, all impacted test cases were identified as obsolete. On the other 11 versions, however, rerunnable test cases were found, and their numbers represented only 3.74% to 12.29% of the entire test suite. Such results suggest that, at least for the types of corrective modifications considered, our test re-use approach can identify a reasonable number of test cases for re-use.

Columns 3, 4, and 9 let us consider the effectiveness of re-used test cases in re-establishing testedness after modifications. In six of the 14 cases considered, test re-execution restored testedness to within 1% of its original level (in three cases final testedness equaled, and in one case it exceeded, initial testedness). In all other cases except those where test cases were obsolete, final testedness fell within 5% of original testedness. This shows that when our test reuse methodology finds rerunnable test cases, those test cases can be effective at restoring testedness in portions of the spreadsheet affected by modifications,
allowing end users to re-achieve much of their initial coverage without creating new test cases.
CHAPTER 5
CONCLUSION

In this thesis, we have described test case generation and reuse methodologies developed for use with the WYSIWYT spreadsheet testing methodology. Our methodologies are user driven and are tightly integrated with the spreadsheet programming environment. When the user requests help from the system, the system responds with a test case, either reused or new, that will help them increase the testedness of the indicated spreadsheet, cell or du-association. For purposes of test case generation, we have used both random and dynamic, goal-oriented algorithms. Test reuse was implemented through algorithms for recording test case information when the user applies a test, and performing impact analysis to determine which of these tests need to be rerun after a change to the spreadsheets. Both of these methodologies were prototyped in Forms/3, and we performed studies to show that we could effectively and efficiently generate new test cases and select test cases for reuse.

The results from the study of automated test case generation suggest the following possible future work:

- In general, the Random technique was faster at reaching low levels of coverage, but was unable to reach levels of coverage as high as those reached by as the Chaining technique. This suggests that some combination of techniques might yield more effective results.
Since our final goal is to help end users, end-user studies are required to assess the ability of end users to effectively use our test case generation methodology.

The results from the study of our test reuse methodology suggest the following work:

- So far we have only considered the ability of our methodology to reuse test cases after simple changes on a single spreadsheet. Further experiments with more complex changes on additional spreadsheets are required before generalizing these results.

- At this time, we have only developed a preliminary interface that does not provide the user with any information about the tests that are being reused. Additions and improvements to the interface could provide test history or other testing information when tests are being reused.

- End-user studies are required to determine whether users can understand and use our test reuse methodology.

It is our hope that through our current and future work, that we can ease the onerous task of testing spreadsheets, and thereby help end users create more accurate and reliable spreadsheets.
BIBLIOGRAPHY


