AN ABSTRACT OF THE THESIS OF

Mingming Cao for the degree of Master of Science in Computer Science presented on June 27, 2000.

Title: Automatic Test Case Generation for Spreadsheets

Abstract approved: Redacted for Privacy

Gregg Rothermel

Test case generation in software testing is a process of developing a set of test data that satisfies a particular test adequacy criterion. It is desirable to automate this process since doing it manually is not only technically difficult but also tedious and time-consuming. Although there has been considerable research in automatic test case generation directed at imperative languages, we find no research exists addressing the problem for spreadsheet languages. This problem is particularly important for spreadsheet languages, since spreadsheet languages are widely used by end users and most of them lack testing backgrounds. To address this need, in this thesis, we present an automatic test case generation methodology for spreadsheet languages. Based on an analysis of the differences between imperative languages and spreadsheet languages, we developed our methodology by properly adapting existing test case generation techniques for imperative languages. Our methodology is integrated with a previously developed methodology for testing spreadsheets, and supports incremental automatic test case generation and visual feedback. We have conducted a family of empirical studies to assess the effectiveness and the efficiency of the essential techniques underlying our methodology. The results of our studies show that the test cases generated by our methodology can exercise a large percentage of a spreadsheet under test. The results also provide insights into the tradeoffs between two test case generation techniques for spreadsheet languages.

Automatic Test Case Generation for Spreadsheets

by

Mingming Cao

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed June 27, 2000
Commencement June 2001

Master of Science thesis of Mingming Cao presented on June 27, 2000

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Chair of the Department of Computer Science

Redacted for Privacy

Dean of the Graduate School

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.

Redacted for Privacy

Mingming Cao, Author

ACKNOWLEDGMENTS

I would like to thank my major professor Dr. Rothermel for his invaluable ideas, helpful suggestions and detailed comments. I appreciate the time and energy he put into this thesis.

My thanks also go to Dr. Burnett for her gracious help, to Dr. Cook for his precious comments and to Dr. Van Der Mars for being on my committee.

My particular appreciation goes to Karen Rothermel, who gave me valuable suggestions in statistical analysis, and Andy Ko, who created the large spreadsheets for the empirical studies. Thanks also to all other members of the Visual Programming Research group who provided help with my implementation.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

AUTOMATIC TEST CASE GENERATION FOR SPREADSHEETS

Chapter 1

INTRODUCTION

Of all the programming languages being used today, spreadsheet languages may be the languages most frequently used by end users. Spreadsheet programs are created by end users to perform a wide variety of tasks, for example, for computing personal taxes, student's grades and employee's salaries. Other research programs from the spreadsheet language paradigm are also designed for a variety of purposes, such as visual matrix manipulation [28], providing steerable simulation environments for scientists [6] and specifying GUI [18].

There is a growing awareness that, despite the perceived simplicity of spreadsheet languages, spreadsheet programs frequently contain faults [4, 20, 21]. Recent studies report that 44% of a set of "finished" spreadsheets still contained faults [4]. One possible factor in this problem is that end users may be overconfident about their spreadsheets. Another possible factor is the absence of help for testing spreadsheets, especially help for end users without testing backgrounds.

The awareness of this inadequacy motivated the designation of a testing methodology for spreadsheet languages (WYSIWYT) [26]. The WYSIWYT (What You See Is What You Test) methodology allows spreadsheet programmers to incrementally try test data, and provides visual feedback about that test data. The visual feedback can guide the programmers in testing their spreadsheets more adequately and more efficiently. Empirical studies show that the methodology can increase end users' testing effectiveness and reduce their overconfidence [27].

This methodology, however, relies solely on the intuitions of spreadsheet programmers to identify test cases to thoroughly test their spreadsheets. The process of identifying test cases is laborious and time-consuming, and its success depends on the experience of the programmer. This problem is more serious for end users of spreadsheet languages since most of them are not experienced programmers and lack background in testing. If the test case generation process for spreadsheets could be automated, the whole spreadsheet testing process could be easier for spreadsheet users. Examining existing work on automatic test case generation, however, we find that most research in this field is directed at imperative languages [2, 3, 8, 9, 11, 13, 14, 15, 16, 17, 19, 22], and no discussion exists addressing automatic test case generation for spreadsheet languages.

To address this lack, in this thesis, we present an automatic test case generation methodology for spreadsheet languages. Based on an analysis of the differences between imperative languages and spreadsheet languages, we developed our methodology by adapting existing techniques for imperative languages. Our methodology is integrated with the WYSIWYT methodology to support incremental automatic test case generation and visual feedback.

The materials presented in this thesis are organized as follows. Chapter 2 provides the necessary background information about spreadsheets and reviews the literature on automatic test case generation techniques for imperative languages. In Chapter 3, we discuss the requirements for an automatic test case generation methodology for spreadsheet languages, then present our methodology based on that discussion. Chapters 4, 5 and 6 present three empirical studies performed to investigate the effectiveness and efficiency of our methodology. Finally, we conclude and discuss future work in Chapter 7.

Chapter 2

BACKGROUND

In this chapter, we present background material about spreadsheet languages and the WYSIWYT testing methodology for spreadsheet languages. We then review automatic test case generation (ATCG) techniques for imperative languages.

2.1 Spreadsheet languages

Spreadsheet languages support a declarative approach to programming [1]. A spreadsheet can be viewed as a collection of cells. Users create a spreadsheet by creating cells and defining formulas for those cells. A cell's value is calculated through its formula, which may reference values of other cells. Once a cell's formula is determined, the underlying evaluation engine automatically calculates that cell's value, and follows data dependencies to calculate the values of affected cells. The updated results are shown immediately.

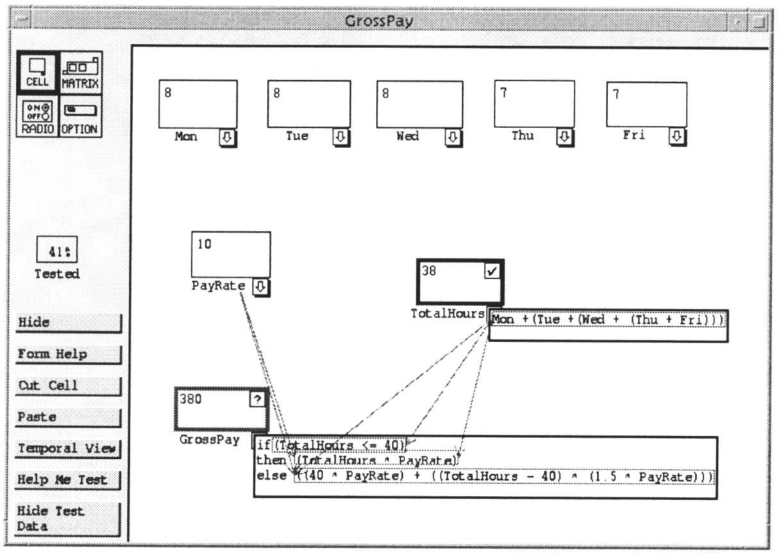
Previous work showed that significant differences exist between spreadsheet languages and traditional imperative languages [25]. There are three major differences. First, evaluation of spreadsheets is driven by data dependencies between cells, and spreadsheets contain only local explicit control flow within each cell. Second, spreadsheets are developed incrementally with immediate visual feedback. Third, imperative languages are mostly used by professional programmers, while spreadsheet languages are used not only by programmers, but more importantly, they are widely used by end users who have no formal software engineering background. These three differences must guide the development of testing and debugging methodologies for spreadsheets.

2.2 WYSIWYT methodology

Attending to these differences between imperative languages and spreadsheet languages, previous research has developed an incremental visual testing methodology "What You See Is What You Test" (WYSIWYT), for testing spreadsheets [24, 26]. The testing process is an incremental process: users incrementally change input cells' values, the underlying engine automatically evaluates cells, and users validate the results displayed in affected output cells. As a user incrementally develops a spreadsheet, he or she also tests that spreadsheet incrementally. A prototype of this testing methodology has been integrated into the spreadsheet language Forms/3 [5], as Figure 2.1 shows. The examples in this document are presented in Forms/3. We overview the WYSIWYT methodology here. More detailed information about it can be found in [24, 25, 26].

2.2.1 User's view

In the WYSIWYT implementation, several types of testing information are provided. First, the color of a cell indicates the extent to which the cell has been tested. Red means untested, blue means fully tested, and shades of purple mean partially tested. Cells start with red borders, and their borders become more blue as those cells are tested more thoroughly. Second, a testedness indicator displays the testedness of the whole spreadsheet under test. Third, each cell



```
Mon +(Tue +(Wed + (Thu + Fri)))
```

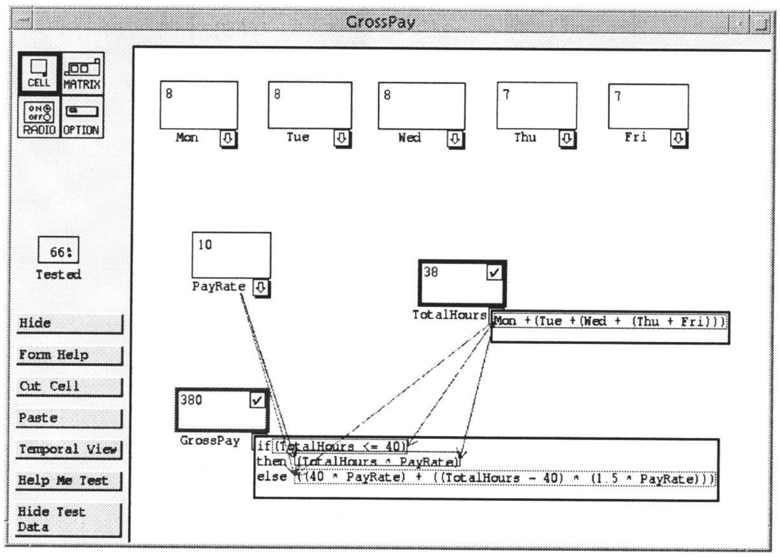
```
if (TotalHours <= 40)
then (TotalHours * PayRate)
else ((40 * PayRate) + ((TotalHours - 40) * (1.5 * PayRate)))
```

FIGURE 2.1: Forms/3 GrossPay with testing information displayed (after the first validation)

contains a checkbox, and user actions change the checkbox status. Three types of checkbox status are possible: empty, question mark, or check mark. Both an empty check box and a question mark indicate that the cell has not been validated under the current inputs. However, the question mark indicates that validating the cell would test something that had not been previously validated, whereas the empty checkbox indicates that validating the cell would not validate anything that had not been previously validated. The check mark occurs in a cell with a question mark after the user validates it, representing a validation based on the current inputs. Fourth, the interactions between cells caused by cell references can be visually viewed by the user by displaying dataflow arrows between cells or subexpressions in cell formulas. These arrows illustrate information about testedness at a finer granularity. Following the same color scheme as for the cell borders, red arrows indicate that the corresponding interactions have not been tested, blue arrows indicate that the corresponding interactions have been tested and purple arrows indicate that the corresponding interactions have been partially tested.

Initially, all output cells start with red borders (untested). Whenever the user notices that the value of a cell under test is correct, she validates it by clicking the checkbox in its right corner. As a result, the underlying engine propagates the implication of the successful test to cells that contribute to it, and reflects this progress in "testedness" by changes of the sort described above.

As an example, consider Figure 2.1, which displays a spreadsheet GrossPay and its testing information. GrossPay calculates the weekly gross pay for a person. The weekday working hours and the pay rate per hour are given in the input cells *Mon* through *Fri* and *PayRate*, the output cell *TotalHours* calculates the total working hours per week and the output cell *GrossPay* calculates the gross pay per week. Suppose the user notices that the value of cell *TotalHours* is correct and validates it. Figure 2.1 shows the feedback after the first validation. The border of cell *TotalHours* is blue (in this figure it is black), indicating that the cell is fully tested. The border of cell *GrossPay* remains red (in this figure it is grey), indicating that the cell is not tested yet. And the testedness indicator indicates 41% parts of this spreadsheet have tested. Suppose the user then determines that the value of cell *GrossPay* is also correct and validates that cell. The effect of the second validation is shown in Figure 2.2. The border color of cell *GrossPay* is changed to purple (deep grey in this figure) and some of the arrows between *TotalHours* and *GrossPay* become blue. The testedness indicator illustrates that 66% of this spreadsheet is tested. To test more adequately, the user must now enter a different test case.



```
Mon +(Tue +(Wed + (Thu + Fri)))
```

```
if (TotalHours <= 40)
then (TotalHours * PayRate)
else ((40 * PayRate) + ((TotalHours - 40) * (1.5 * PayRate)))
```

FIGURE 2.2: Forms/3 GrossPay with testing information displayed (after the second validation)

2.2.2 Behind the scenes

Although the users of the WYSIWYT methodology are not aware of it, they are actually using a *definition-use* test adequacy (du adequacy) criterion, which is adapted from the *output-influencing-all-du-pairs* dataflow adequacy criterion defined for imperative programs [10]. A test adequacy criterion help testers select test data and measure whether a program has been tested "enough". Testing a spreadsheet under the du adequacy criterion focuses on the *definition-use pairs* (du pairs) in the spreadsheet, and the testedness is calculated by the number of du pairs that have been tested divided by the total number of du pairs in the spreadsheet [26]. A definition-use pair connects an expression in a cell formula that defines a cell's value with expressions in other cells that use the defined cell. There are two types of du pairs: definition-c-use pairs, in which the use expression is a computation expression and the use of the definition is for computation purposes, and definition-p-use pairs, in which the use expression is a predicate expression and the use of the definition is for conditional control purposes.

The du adequacy criterion is defined based on an abstract model of spreadsheets, instead of on the code itself [25]. The abstract model is called a *cell relation graph* (CRG), and depicts the control flow within a spreadsheet's formulas and the data dependencies between its cells. A cell's control flow is represented by a *cell formula graph* (CFG), which is similar to the control flow graph used to represent procedures in imperative programs. Each cell formula graph is a directed graph, in which each node represents an expression in a cell formula and each edge represents the flow of control between expressions. There are three types of nodes contained in a CFG: unique entry and exit nodes, representing initiation and termination of the evaluation of the formula respectively; definition nodes, representing simple expressions in cell formulas that define the value of a cell; and predicate nodes, representing predicate expressions in cell formulas. Two edges are outgoing from a predicate node: one represents the true branch of the predicate expression, and another represents the false branch.

Figure 2.3 shows the cell formula graphs for spreadsheet GrossPay. In this figure, a cell is represented by a dashed rectangle with a corresponding CFG displayed in it. Circle nodes represent either entry nodes or exit nodes, rectangle nodes represent predicate nodes and oval nodes represent definition nodes. The dashed arrows between cells illustrate the data dependencies between them.

The CRG is used to define applicable node and edge adequacy criteria for spreadsheets. A node n is exercised by a test case t when the corresponding expression is evaluated under t; a du pair is exercised by t when both its definition node and use node are exercised by t. A test suite is du adequate for

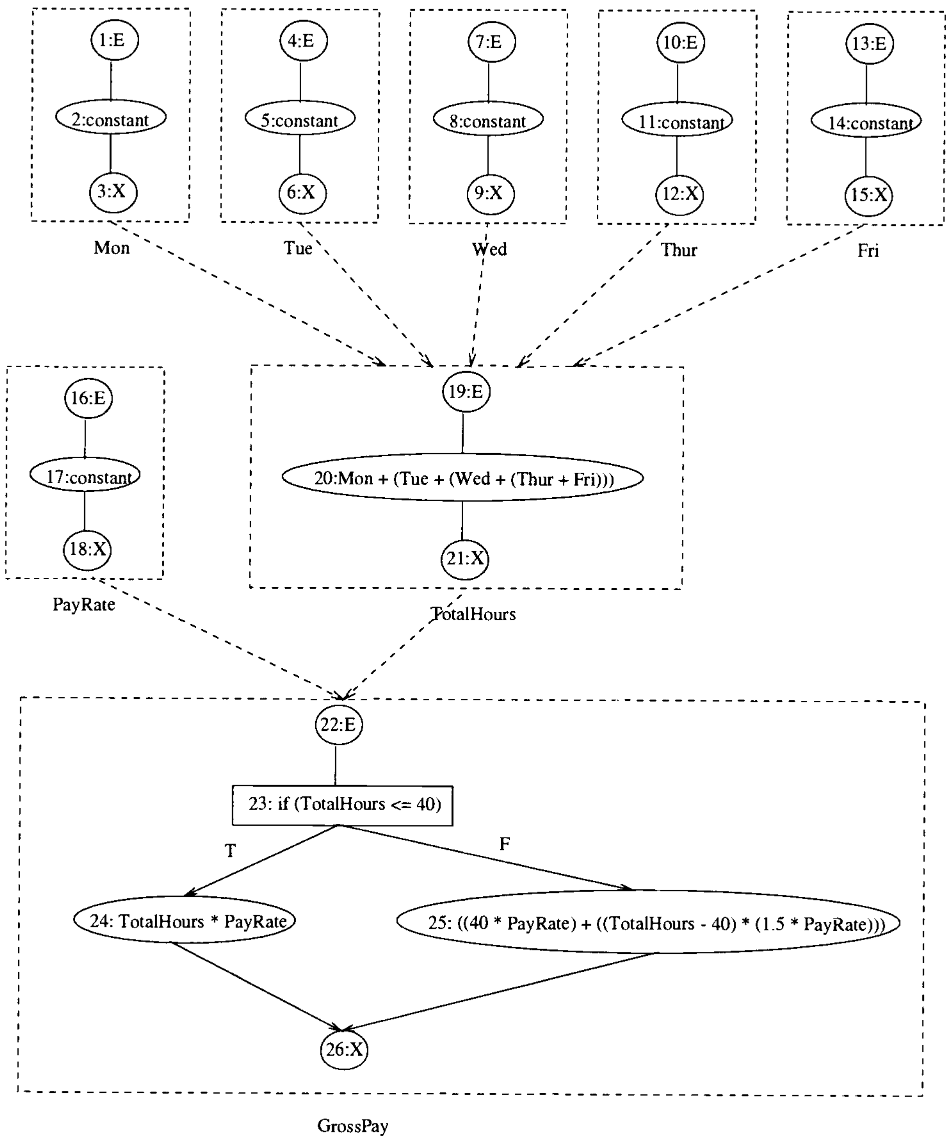


FIGURE 2.3: Cell Relation Graph for GrossPay

a spreadsheet if each definition node and use node pair in the spreadsheet is exercised by at least one test case in that test suite.

However, it is not always possible to exercise all du pairs. Those du pairs that can not be exercised by any inputs are called *infeasible du pairs*. Some infeasible du pairs are caused by contradictory conditions. Infeasible du pairs frequently occur in practice. However, identifying infeasible du pairs is impossible in general [12, 29].

Du pairs in a spreadsheet are visually represented by the arrows between definition expressions and use expressions. The color scheme on the arrows lets a user pay attention to the untested arrows (du pairs); this helps the user identify "useful" test cases to exercise the untested du pairs [23].

2.3 ATCG techniques for imperative languages

There has been much discussion in the literature about techniques for automatic test case generation for imperative languages. Depending on what aspect of these techniques is to be addressed, there are different ways to classify these techniques [30]. An ATCG technique could be categorized according to the test adequacy criterion it uses, for example, statement coverage or branch coverage. ATCG techniques could also be classified according to the goal of the technique. For example, some techniques generate test cases to exercise particular paths, and are called *path-oriented* techniques. Other techniques assume a goal of exercising a particular statement regardless of path, and are called *goal-oriented.*

Another important criterion involves how to generate test cases. There are three types of ATCG techniques according to this criterion: *random* test case generation techniques [3], *static* test case generation techniques [7, 8, 9, 13, 19] and *dynamic* test case generation techniques [11, 14, 15, 16, 17]. Briefly, random techniques generate test cases by randomly selecting input values, static techniques generate test cases by symbolically (statically) executing the program under test, and dynamic techniques generate test cases through dynamic execution of the program under test.

Static and dynamic approaches are the most widespread approaches that have been proposed. They are both based on the code of a program, instead of its specification. Much research work has been done in these two areas, indicating that they are fundamentally different.

Static approaches generate test cases through symbolic walking of a program and creation of a set of constraints required to meet an adequacy criterion. More precisely, such techniques select a path to meet the adequacy requirement first, then find the constraints, in terms of input variables, required to execute that path by symbolically executing the programs, and finally attempt to solve these constraints and obtain a solution. The values that are located in that solution are the test case generated to meet that test adequacy requirement. One advantage of static methods is that infeasible du pairs can often be identified through the use of a constraint solver. However, static methods require high memory to store the expressions encountered during symbolic execution, and require a powerful constraint solver to solve complex equalities and inequalities. In addition, static methods have other problems in dealing with arrays, aliases, loops and complex expressions.

In contrast, dynamic approaches actually execute programs, and take advantage of information obtained during execution to guide the search for a test case that meets the requirement. Dynamic methods reduce the problem of test case generation to a sequence of subgoals, using function minimization to solve these subgoals. Execution-oriented methods [17] and goal-oriented methods [15] are two typical dynamic methods. Both are based on program execution, dynamic data flow analysis, and function minimization methods; however, the execution-oriented methods focus on an execution path while the goal-oriented methods focus on the final goal and ignore the path. Compared to static methods, dynamic methods can take advantage of the actual variable values obtained during execution to try to solve problems with arrays, aliases, loops and complex expressions. In addition, dynamic methods require smaller memory and do not require a complex inequalities and equalities solver for constraints. However, dynamic methods are not good at identifying infeasible du pairs.

Chapter 3

AUTOMATIC TEST CASE GENERATION FOR SPREADSHEETS

Developing test cases is a tedious and time-consuming process in software testing. This can be especially true for end users testing spreadsheets, since end users typically have no testing background. Automatic test case generation (ATCG) techniques assist users in finding test cases; however, existing ATCG techniques are all designed for imperative languages. To address this problem, in this chapter, we present an ATCG methodology developed for spreadsheets.

3.1 Requirements for an ATCG methodology for spreadsheets

As for imperative languages, ATCG methodologies for spreadsheet languages must accept a program (spreadsheet) and a test adequacy criterion as input, and then automatically output test cases that meet the criterion. However, there are other considerations for ATCG for spreadsheets that are different from those for imperative languages. In previous chapters, we described the main differences between spreadsheet languages and traditional imperative languages, as well as the WYSIWYT testing methodology for spreadsheet languages. To be integrated with the WYSIWYT testing environment and accommodate these differences, an ATCG methodology for spreadsheets should satisfy the following requirements:

- *Code-based ATCG.* Since many spreadsheet end users are not experienced programmers, few of them are likely to create specifications for their spreadsheets. Thus, the ATCG methodology for spreadsheets should focus on code-based test case generation techniques.

- *Use a data dependency test adequacy criterion.* Evaluation of spreadsheets is driven by data dependencies between cells. As a result, spreadsheets contain no single explicit global control flow graph, although they contain local explicit control flow within each cell. This suggests that an ATCG methodology must be compatible with the data dependency driven evaluation model and not rely upon any particular evaluation order. To be integrated with the WYSIWYT methodology, the ATCG methodology for spreadsheets is required to use du adequacy as its test adequacy criteria.

- *Incrementally generate test cases and provide visual feedback.* Spreadsheet languages let users program incrementally. This feature suggests that ATCG methodologies should support incremental test case generation. Upon the end user's request, an ATCG technique should generate one test case that can increase the cumulative testedness. Following completion of this task, the system should let the user continue their activities, which may include requesting additional test generation help.

  In addition, spreadsheet programming environments are also characterized by visual feedback. Thus, ATCG methodologies should also provide visual feedback. For instance, if a user requests help in finding a test case for a spreadsheet, the ATCG subsystem should generate a test case as requested, and gives visual feedback by updating the input cells' values and indicating where validation may be helpful.

Moreover, since spreadsheet programming environments consist of a visual interface subsytem and a underlying evaluation engine subsystem, ATCG methodologies for visual programming languages should also be divided into two subsystems: the visual interface part, which accept users' requests and gives visual feedback, and the underlying ATCG technique subsystem, which generates test cases upon request. Users should communicate with the visual interface part and should not need to understand the underlying ATCG techniques. This would allow ATCG methodologies to provide multiple ATCG techniques and combine them as necessary without involving the users in the mechanism that combines them.

- *Generate test cases at the whole spreadsheet level.* This is the basic functionality that an ATCG methodology should provide for spreadsheet testing. When requested by the user, the ATCG should assist users in finding a test case, which executes an untested part of the spreadsheet.

- *Generate test cases at the cell level.* ATCG methodologies should also assist users in generating a test case for a particular cell in a spreadsheet. This situation may occur when a user finds some cell not well tested and does not wish to manually identify test cases that improve the testedness of that cell. By selecting the cell of interest, users can ask for help from the ATCG subsystem. To satisfy a user's specific requirement, an ATCG subsystem can focus on the specified cell and attempt to find a test case that exercises an untested du pair whose coverage will increase the testedness of that cell.

- *Generate test cases at the du pair level.* Since the underlying adequacy criterion used in our ATCG methodology is definition-use adequacy, the final goal of testing a spreadsheet is to find test cases that exercise all the du pairs in the spreadsheet. During the testing process, one or more du pairs may particularly interest users. ATCG methodologies should provide the functionality to generate a test case at the du pair level. The WYSIWYT methodology supports selecting du pairs by letting users click arrows between the source of a du pair and the destination of that du pair. When users select one or more arrows, our ATCG methodology must attempt to generate test cases to cover the associated du pairs.

- *Provide additional test cases on request.* The WYSIWYT methodology is an incremental testing methodology. The whole testing process is composed of iteratively finding a test case and validating output cells. When users are not able to judge whether the outputs associated with a test case are correct, they may need additional test cases to support their decision. In that case, ATCG should attempt to find a test case that differs from the current one but exercises the same part of the spreadsheet that the users are interested in.

The requirements we have just discussed are particular requirements for spreadsheet ATCG methodologies. However, there are some features of spreadsheet languages that may make the design of ATCG techniques easier than for imperative languages:

- *Incremental evaluation.* In essence, the evaluation strategies used in spreadsheet languages are either eager or lazy. Eager evaluation is driven by formula changes: whenever a cell's formula is changed, the change is propagated to every cell that is affected by this change. Only the affected cells are recalculated and other cells that are not affected retain their values. Lazy evaluation is driven by output requests: whenever a cell c is required to calculate its value, it is computed and so is every cell that cell c needs. Lazy evaluation computes fewer cells than eager evaluation does. Both evaluation strategies incrementally execute part of the program. Compared to the complete evaluation strategy used in many imperative languages, this incremental evaluation feature of spreadsheets may improve the efficiency of ATCG techniques that require the dynamic execution of programs.

- *Absence of redefinitions.* In spreadsheet languages, cells act as variables, and the value for cell c can be defined only in its formula. Exercising a definition node and a use node in a du-pair during one execution guarantees the coverage of that du pair. However, this is not the case in traditional imperative programs, because a variable in an imperative program could be re-defined during its lifetime. Even when both the definition node and the use node in an imperative program are exercised during one execution, the du pair connecting them cannot be guaranteed to be exercised, since the variable involved in the du pair may be re-defined by another statement located on the execution path between the definition node and the use node. To determine whether a du pair is exercised, in addition to checking the definition node and the use node of that du pair, the ATCG techniques for imperative languages must check the nodes on the execution path between those nodes. This procedure is simplified in ATCG for spreadsheets due to the absence of redefinitions.

Our goal was to develop an ATCG methodology for spreadsheet languages that meets the general requirements for spreadsheet languages, and takes advantages of the features listed above. The key to developing the overall ATCG methodology for spreadsheets is the underlying ATCG technique. Given the large body of research on ATCG techniques for imperative programs, however, a natural first step was to see whether such a technique could be obtained by adapting an appropriate existing technique. The question is, among the three types of ATCG techniques for imperative languages described in Chapter 2, which might be most appropriate?

As we described in Chapter 2, depending on the requirement for a test case, static ATCG techniques build a set of constraints to meet that requirement through symbolic execution. Dynamic ATCG techniques take advantage of the actual execution, searching for a test case under the guide of a local optimization method. Examining these two types of ATCG techniques, we find that static ATCG techniques, although not requiring actual spreadsheet execution, would require a constraint solver which could solve complex equalities and inequalities in terms of the input variables. Also, although static ATCG techniques may be better than other techniques at identifying infeasible du pairs, they have higher storage requirements because they must store the expressions calculated during the symbolic execution.

Consideration of these advantages and disadvantages favors dynamic techniques. Considering specific dynamic techniques, the goal-oriented technique is of particular interest for several reasons:

- We can view the testing process in the WYSIWYT testing methodology as a process of meeting a sequence of subgoals, each involving exercising a particular unvalidated du pair. Each subgoal can be achieved by exercising the definition node of the du pair and the use node of that du pair, regardless of paths between them. The goal-oriented technique fits naturally with this view.

- The goal-oriented technique is a dynamic technique, requiring actual execution of the program under test. In addition, to search for a test case, it requires re-executing of the program to evaluate the branch function once new test data is tried. Thus, evaluating the branch becomes the most time-consuming part of the test case generation. This could cause an efficiency problem especially when the program under test is large and there are many input variables. This problem is reduced in spreadsheet languages that support the "incremental evaluation" feature — whenever an input cell's value is changed, the engine could evaluate only the cells that are concerned. In this way, the amount of evaluation is kept at a minimum.

- Another requirement of the goal-oriented technique is instrumentation to monitor execution traces. This becomes easy to satisfy within the WYSIWYT testing methodology, because it is already taken care of when the WYSIWYT methodology fulfills its validation functionality.

- Korel mentions that searching only on input cells that affect the branch function could speed up the search procedure in the goal-oriented technique [17]. Since the WYSIWYT system provides backward and forward data dependence analysis, identifying the input cells that affect the branch function is easy.

It seems that applying the goal-oriented ATCG techniques to spreadsheets with the WYSIWYT testing methodology requires less effort than doing so in imperative languages, and will be less expensive than applying static ATCG techniques. These reasons persuaded us to use the goal-oriented technique. We will discuss our adaptation of this technique to spreadsheets in Section 3.3.

Although both static and dynamic ATCG techniques could generate test cases for a test adequacy criterion, the generation processes are complex. A simple alternative is a random ATCG technique. A typical random technique randomly selects values for inputs. Compared to other techniques, the random technique has not been considered "intelligent" enough to use on imperative programs when the goal is to satisfy a test adequacy criterion. However, random generation has some advantages: it requires much less effort to implement and has less calculation overhead than static or dynamic techniques. Also, it seems possible that the difficulties of randomly generating inputs to obtain specific coverage may be fewer for spreadsheets than for imperative programs. Thus, we designed an ATCG technique that generates test cases randomly but under the guidance of a test adequacy criterion. We will discuss this technique in detail in Section 3.2.

3.2 Random technique

The random ATCG technique we developed for spreadsheets is similar to the procedure used in random testing. However, our approach adds considerations of test adequacy to this random generation. In other words, our random technique attempts to generate test cases to satisfy the du adequacy criterion. We describe the algorithm for random test case generation in Figure 3.1. In this algorithm, a set of input values V is randomly selected initially, then they are assigned to the input cells set I in spreadsheet S to force S to execute automatically. After that, the algorithm determines all the du pairs exercised by this execution. If there is one or more du pair in a *candidate du pairs list* (a list of du pairs that are not yet validated) that are exercised by these values, these values are together considered a test case and the algorithm terminates. Otherwise the algorithm tries other random values and repeats this process until a test case is found or time is exceeded.

```
Algorithm RandomGen
Input: Subject spreadsheet $S$ with input cells $I = \{i_1, i_2, \ldots, i_k\}$, and ranges $R = \{r_1, r_2, \ldots, r_k\}$
Output: Test case $T$
1.begin
2.     collect all unvalidated du pairs in $S$ into candidate du pairs list dulist
3.     while not time out
4.         randomly generate input values $V = \{v_1, v_2, \ldots, v_k\}$ within the provided ranges $R$
5.         assign each $v_j$ to its corresponding input cell $i_j$
6.         re-evaluate cells that are sinks of one or more du pairs in dulist
7.         if there are some unvalidated du pairs in dulist exercised by inputs $V$
8.           return V
9.         endif
10.    endwhile
11.    return NIL /* no test case found*/
12. end
```

FIGURE 3.1: Algorithm for random ATCG technique for a whole spreadsheet

The algorithm in Figure 3.1 depicts the random test generation procedure at the whole spreadsheet level. Algorithms for generating test cases at the cell level and at the du pair level are similar to the one at the whole spreadsheet level. The difference lies only in the candidate du pairs list: at the whole spreadsheet level, the candidate du pairs list contains all the unvalidated du pairs in the spreadsheet under test; at the cell level, only the unvalidated du pairs whose uses are contained in the selected cells are included in the candidate du pairs list; at the du pair level, only the required du pairs are considered.

To avoid the screen flush caused by updating dependencies associated with an unsuccessful try, the random generation is designed to be an "invisible" calculation procedure to the end user. Before the random ATCG has found a test case, all cells keep their original values. Only after a test case is generated will the end user see the updated value of the input cells changed and all the cells affected by this test case updated. If the technique fails to find a test case and terminates, all the values remain the same as when the generation started.

3.3 Goal-oriented technique

The goal-oriented technique, as a dynamic ATCG technique, actually executes the program under test. It transforms the goal of reaching a particular node into a sequence of subgoals, which aim to reach a particular branch node necessary to reach the final goal. Based on dynamic execution, it uses function minimization techniques and data flow analysis to search for input values that satisfy each subgoal.

This section presents our adaptation of the goal-oriented technique for spreadsheets. In the following subsections, we first describe the whole procedure and tasks involved in the goal-oriented technique, then discuss in detail about how we design our goal-oriented technique for spreadsheets to fulfill those tasks.

3.3.1 Generation procedure

Since the nature of the goal-oriented technique is to generate a test case for a particular goal, we discuss our goal-oriented technique at the du pair level first.

The goal of test case generation under the WYSIWYT methodology can be seen as one of exercising two nodes, the definition node of a particular du pair and the use node of that du pair. The whole generation procedure in the goal-oriented technique for a particular du pair is similar to the one for imperative programs. We begin by outlining the procedure, then provide details:

1. Identify the two subgoals — the definition node and the use node of a particular du pair — and the *constraint path* — the sequence of nodes that must be exercised to achieve the goal. [1]

2. Starting from the current inputs, request the spreadsheet evaluation engine to evaluate the cell that contains the use node and collect the execution traces for the definition cell and the use cell.

3. Compare the constraint path with the current execution traces. If an unexpected node occurs in the execution traces, determine the desired branch associated with that unexpected node from the execution sequence, and associate it with a real-valued function, called a *branch function*,

[1] The "constraint path" is not the same set of constraints used in static ATCG techniques, which is a set of inequalities and equalities written in terms of input cells.

and the node at which the branch starts, called a *break node* for this generation. The break node, the branch out of this break node that we want to cover, and the branch function associated with the desired branch, are encapsulated in an abstract object called a *break point.*

4. With the branch function of the desired branch, use a function minimization algorithm to attempt to locate input values that would cause the execution flow to follow the previous constraints and exercise the desired branch.

5. Repeat steps 2 to 4 iteratively, solving subgoals one by one until there is no unexpected node appearing in the execution traces, which means the definition node and the use node have been exercised.

If the above process terminates with output prior to time-out, the input values give the test case generated to exercise the desired du pair.

We describe the generation algorithm in Figure 3.2. In this algorithm, several procedures are called: *getConstraintPath* returns the constraint CP that must be satisfied to reach a given node; *getBreakPoint* returns the first break point $BPoint$ according to the constraint CP and the execution trace *exeTrace*; *coverBreakPoint* attempts to change input values to force the execution flow to exercise the desired branch. These procedures will be discussed in detail in the following subsections. More precisely, this algorithm starts by obtaining the constraint path that must be traversed to exercise the given du pair; then analyzes the execution trace and the constraint path, acquiring the first break point if there are any; after that, it calls *coverBreakPoint* to solve this break point. The last two steps are executed alternately in this algorithm until the desired du pair is exercised.

```
Algorithm GoalGenAPair
Input: A du pair P
Output: Test case T
1.begin
2.    CP = getConstraintPath (P.definitionNode) + getConstraintPath (P.useNode)
3.    exeTrace = getTrace (P.definitionCell) + getTrace (P.useCell)
4.    while not time out
5.        if P is exercised
6.            return current input values
7.        else
8.            BPoint = getBreakPoint ( exeTrace, CP )
9.            coverBreakPoint ( BPoint )
10.           if inputs to cover the desired branch are not found
11.               return NIL /* no test case is generated */
12.           endif
13.       endif
14.   endwhile
15.   return NIL /* no test case is generated */
16. end
```

FIGURE 3.2: Algorithm for goal-oriented ATCG technique for a du pair

Exercising any one of the unvalidated du pairs in a spreadsheet or a selected cell could be considered the goal of test case generation at the whole spreadsheet level or at the cell level. Thus, as in the random technique, the generation procedures in the goal-oriented technique at different levels differ only in terms of a "candidate du pairs list". To generate a test case, the goal-oriented technique starts from the first du pair p of the candidate du pairs list *duList*. It will return a test case if during the course of searching test case for du pair p it covers *p* or other pairs in *duList*. If it fails to find a test case, it will put *p* on the end of *duList* and select the next du pair as a subgoal. We generalize the goal-oriented generation procedure to the three levels in Figure 3.3.

```
Algorithm GoalGen
Input: A testable object TO
Output: Test case T
1.begin
2.     collect all unvalidated du pairs into candidate du pairs list duList
3.     while not time out
4.         remove du pair P from the head of duList
5.         TC = GoalGenAPair ( P )
7.         if success
8.           return TC
9.         else append P to the end of duList
9.         endif
10.    endwhile
11.   return NIL /* no test case is generated*/
12. end
```

FIGURE 3.3: Algorithm for goal-oriented ATCG technique at all three levels

Overall, the kernel of the generation procedure in the goal-oriented technique for a du pair is the three subprocedures: obtaining the break point on the execution trace, creating a branch function for that break point, and searching input values under the guidance of the branch function to exercise the desired branch. We will discuss in detail how these three subprocedures function in our technique in the next subsections. We present the material in the context of the Forms/3 environment and the WYSIWYT methodology; however, the technique could be implemented for other spreadsheet languages by appropriate substitutions.

3.3.2 Task1: obtaining the break point

A break point is an abstract object that maintains the information associated with a branch during goal-oriented generation. It contains the break node, a desired branch starting from that break node, and the branch function associated with the desired branch. This subsection describes how our technique detects the break node and determines the desired branch.

To detect the break point, constraint path information and execution trace information are necessary. The execution trace information for a cell is available under the WYSIWYT methodology, since each cell has a tracer to monitor its execution flow. The whole execution trace concerned with a du pair is built by concatenating the execution trace for the definition cell and the execution trace for the use cell.

To obtain the constraint path associated with a particular du pair, we also utilize some functionality provided in the WYSIWYT methodology. As we described in Chapter 2, each cell has a cell formula graph (CFG). Starting from the first node in a CFG, we can reach all the other nodes in that CFG. Thus, in our goal-oriented ATCG technique, to obtain the constraint path for the definition node, we find the CFG that the definition node belongs to, start from the first node in that CFG, and recursively enumerate the path from there to the definition node. In the constraint path for the definition, the definition node is also included, together with the predecessor of that node, for the purpose of identifying which branch is desired. In the same way as with the definition node, we also obtain the constraint path for the use node in the du pair. In this way, we construct the constraint path for the entire selected du pair. Figure 3.4 depicts the algorithm for finding the constraint path for a node.

Having the constraint path and the execution trace for a du pair, we obtain the first break node and desired branch by pattern search. The algorithm used to detect a break point is shown in Figure 3.5.

After obtaining the break node and the desired branch, we build the branch function associated with it. We will discuss how to build a branch function next.

3.3.3 Task2: creating branch functions

The branch function plays an important role in the search to find a solution for a subgoal, i.e. exercising the desired branch. It is a real-valued function that is created by transforming the desired branch's predicate expression into an arithmetic expression. Based on different operators in predicate expressions and desired branches, there are different ways to build a branch function. However, to fulfill its role in the searching procedure and be sufficiently general, the branch function designed for a particular type of predicate expression should have the following properties:

Property 1: As a guide in the searching procedure, changes in the values of the branch function should reflect changes in closeness to the goal. For example, if the current input values are closer to the solution of the goal than the previous ones, the current value of the branch function should be greater than the value associated with the previous input values.

```
Algorithm getConstraintPath
Input: the goal CFG node gnode
Output: a list of CFG nodes CP
1.begin
2.    obtain the parent CFG of gnode, and the first node fnode belonging to that CFG
3.    CP = enumeratePath ( fnode, gnode)
4.    return CP
5.end
Procedure enumeratePath
Input: the current node being enumerated, cnode, the goal CFG node, gnode
Output: a list of CFG nodes CP
1.begin
2.    let CP be empty
3.    if cnode equals gnode
4.       insert cnode onto the head of CP
5.    else if  cnode is a predicate node
6.           CP = enumeratePath ( cnode.truebranchnode, gnode)
7.           if  CP is not empty
8.              insert cnode.truebranchnode onto the head of CP
9.           else
10.                CP = enumeratePath ( cnode.falsebranchnode, gnode)
11.                if  CP is not empty
12.                   insert cnode.falsebranchnode onto the head of CP
13.                endif
14.          else if
15.    else if
16.    return CP
17. end
```

FIGURE 3.4: Algorithm for obtaining the constraint path to a node

```
Algorithm getBreakPoint
Input: A execution path exePath, the constraint path CP
Output: A break point object BPoint
1.begin
2.    let BPoint be NIL, let N be NIL
3.    find the first node N that is in CP but not in exePath
4.    if such an N exists
5.        let BNode be the node just before N in CP
6.        let DBranch be the branch connecting BNode and N
7.        BPoint.breakNode = BNode, BPoint.desiredBranch = DBranch
8.        BPoint.branchFunc = getBranchFunc ( BNode, DBranch )
9.    endif
10.   return BPoint /*returns NIL if no break point*/
11. end
```

FIGURE 3.5: Algorithm for obtaining the break point for a desired du pair

Property 2: The rule used to judge whether a branch function is improved should be consistent across all branch functions, and the rules used to determine whether a desired branch is exercised should be similar to one another. For example, if rule one is "if the value of the branch function is increased, then we are closer to the subgoal" and rule two is "if the value is negative, then the desired branch is not exercised", then when new test data causes the value of the branch function to change from -5 to -2, this indicates that the desired branch is not exercised by this new test data but it is closer to the subgoal.

Considering the two properties listed above, we first identify the general criteria applicable to all branch functions:

1. **Criterion 1:** if the value of the branch function is negative, the desired branch is not exercised;

2. **Criterion 2:** if it is positive (or equal to 0 in some cases), the desired branch is exercised;

3. **Criterion 3:** if the value of the branch function is increased, but still negative, the search that caused this change is considered a successful search.

These criteria are similar to those used by the goal-oriented method presented in [15, 17] except that in that work the roles of negative and positive are reversed.

Based on these criteria, we can design a branch function for each operator and branch combination. The branch predicate expressions defined in Forms/3 are of the following form:

$$E_1 op E_2$$

where E_1 and E_2 could be arithmetic expressions or predicate expressions, and *op* is one of the relational operators $>, \geq, <, \leq, =$, or the boolean operators *and*, *or*, *not* defined in Forms/3. The desired branch could only be *True Branch* or *False Branch*; since switch statements are simply sugar for nested "if"s, supporting them would not require substantive change to the ATCG techniques.

It is relatively easy to create branch functions for predicate expressions containing only relational operators. Based on the three criteria, we designed these branch functions in a manner similar to that used by Korel for imperative programs [17]. The branch functions for predicate expressions containing only relational operators are shown in Table 3.1, where E_1 and E_2 are arithmetic expressions.

Predicate Expressions	True Branch	False Branch	satisfaction condition
$E_1 > E_2$	$F = E_1 - E_2$	$F = E_2 - E_1$	$F > 0$
$E_1 \geq E_2$	$F = E_1 - E_2$	$F = E_2 - E_1$	$F \geq 0$
$E_1 < E_2$	$F = E_2 - E_1$	$F = E_1 - E_2$	$F > 0$
$E_1 \leq E_2$	$F = E_2 - E_1$	$F = E_1 - E_2$	$F \geq 0$
$E_1 = E_2$	$F = -abs(E_1 - E_2)$	$F = abs(E_1 - E2)$	$F = 0$ (true) $F > 0$ (false)

TABLE 3.1: Rules for creating branch functions for predicate expressions containing only relational operators

To illustrate, suppose the break node is a predicate node in cell c and the predicate expression is "$if(a > b)$". Assume the desired branch is the True Branch. Looking in Table 3.1, we obtain the branch function $f(a > b, true) = a - b$. Suppose cell a's value is 3 and cell b's value remains 5, then the value of this branch function is -2. If cell a's value is changed to 7 and cell b's value is 5, then the value of this branch function is increased to 2, which meets the satisfaction condition and indicates that the desired branch is exercised.

Next we discuss how to create branch functions for more complex predicate expressions including relational operators and boolean operators. Korel addresses only the relational operator branch functions in [15, 17], and does not discuss how to deal with the boolean operators. Since the boolean operators occur frequently in spreadsheets, we address this absence in our goal-oriented technique.

Designing branch functions for predicate expressions containing boolean operators is not as straightforward as for those containing only relational operators.

Predicate Expressions	True Branch	False Branch

E_1 *and* E_2	if $(f(E_1, true) < 0)$ *and* $(f(E_2, true) < 0)$
	then $F = f(E_1, true) + f(E_2, true)$
	else $F = min\{f(E_1, true), f(E_2, true)\}$
		if $(f(E_1, false) < 0)$ *and* $(f(E_1, false) < 0)$
		then $F = f(E_1, false) + f(E_2, false)$
		else $F = max\{f(E_1, false), f(E_2, alse)\}$

E_1 *or* E_2	if $(f(E_1, true) < 0)$ *and* $(f(E_2, true) < 0)$
	then $F = f(E_1, true) + f(E_2, true)$
	else $F = max\{f(E_1, true), f(E_2, true)\}$
		if $(f(E_1, false) < 0)$ *and* $(f(E_1, false) < 0)$
		then $F = f(E_1, false) + f(E_2, false)$
		else $F = min\{f(E_1, false), f(E_2, alse)\}$

not E_1	$F = -f(E_1, true)$	$F = -f(E_1, false)$

TABLE 3.2: Rules for creating branch functions for predicate expressions containing boolean operators

However, no matter how complex a predicate expression is, the branch function associated with it must have the two properties listed above. In addition, since the values of the subexpressions being operated on by the boolean operators affect the value of the whole expression, instead of creating a totally new branch function for expressions containing boolean operators, we build branch functions based on the branch functions for the subexpressions.

The rules for creating branch functions for the three kind of boolean operators *and*, *or*, and *not* upon two desired branches *TrueBranch* and *FalseBranch* are described in Table 3.2. In this table, E_1 and E_2 could be any predicate expressions (including an expression containing boolean operators); f is a function which accepts an expression and the desired branch as input and then outputs the branch function for the desired branch of the given expression. For example, $f(E_1, true)$ represents the branch function for the true branch of E_1.

Among the three boolean operators, designing the branch function for the *not* operator is easiest. According to *not*'s logical meaning, we simply build the branch function for the desired branch by adding a negative sign before the branch function for the subexpression with the same desired branch. More precisely, if the goal is to exercise the true branch of predicate *not* E_1, then the branch function is the negative branch function of the true branch of E_1; similarly, if the goal is to exercise the false branch, then the branch function is the negative branch function of the false branch of E_1. Thus, whenever the subexpression is true (the value of its true branch function is greater than 0), then the value of the true branch function for the whole expression is less than 0, which means the true branch of the expression with the *not* operator is not exercised. For example, if the predicate expression is "*not* $(a > b)$" and the goal is to exercise the true branch, the branch function F for this goal is $f(not\ (a > b), true) = -f(a > b, true) = -(a - b)$. Assuming cell a's value is 7 and cell b's value is 5, then the value of the branch function for the true branch of expression "*not* $(a > b)$" is -2.

The rules for *and* and *or* operators are more complex than those for the *not* operator. Since there are two subexpressions with these operators, based on the logical meaning of those operators, we reduce the goal of a desired branch into one or both of the subgoals. For example, to exercise the false branch of an *and* expression, we could exercise either the false branch of the sub expression on the left side, or the false branch of the sub expression on the right side. In some cases, the goal will be achieved only if both subgoals are achieved, while in other cases, achieving either one of the subgoals will achieve the final goal. In addition, the subgoals might not be achieved at the same time. Based on the three criteria, we design branch functions for these operators according to the following rules:

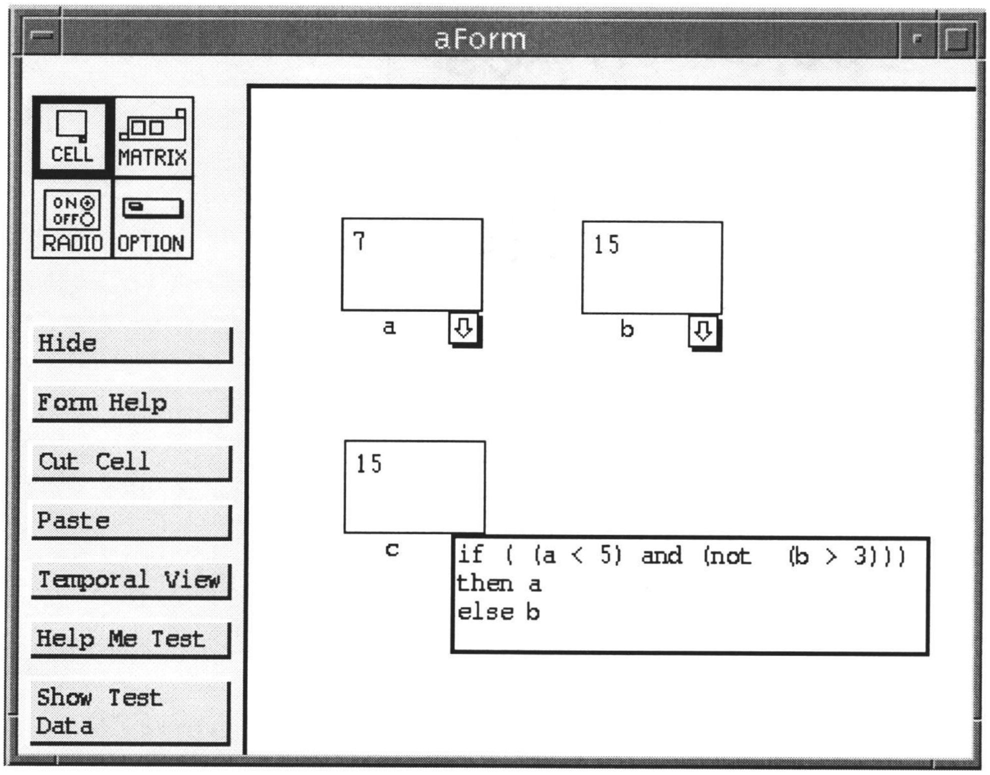
- If neither of the two subgoals is satisfied, then any effort toward either one is considered positive, whether or not both subgoals are required to be met. So in this case the branch function is represented as the sum of the two sub-branch functions. For example, if the two subexpressions are false and the expression contains an *and* operator and we want to exercise the true branch, then the true branch function is the sum of the two true sub-branch functions. Any improvement in either of the true sub-branch functions will increase the value of the whole branch function.

- Obviously, if both the subgoals are achieved, the goal is achieved. In this case, the branch function could be either of the sub-branch functions, since both of them are non-negative. We use the one with the smaller branch function value.

- If one of the two subgoals is achieved while the other is not, then the final goal may or may not be achieved, depending on the number of subgoals required. In the case where only one subgoal is required, the branch function is the branch function of the subgoal that is achieved. In other words, the branch function is the sub-branch function with the non-negative value, thus the branch function for the final goal is also positive. For example, when we want to exercise the true branch of the *or* expression, and only the first subexpression is true, then the branch function is the sub-branch function of that subexpression. In the case where two subgoals are required to achieve the final goal, the branch function for the final goal takes the sub branch function that is not achieved (the one with the smaller branch function value). The reason for this is that we want to focus on the subgoal that is required but not achieved yet, so increasing the value of the branch function for the subgoal already met will not be achieving the final goal.

The general rule is, if the final goal is not achieved, then we focus on the subgoal that has not been achieved yet.

Now we use an example to illustrate how we use the rules in Table 3.1 and Table 3.2 to create branch functions. Figure 3.6 shows part of a spreadsheet, including cells a and b which are input cells, and cell c which is an output cell. Currently cell a's value is 7 and cell b's value is 15. Suppose in the process of exercising a particular du pair, we break at the predicate in cell c and we want to exercise the true branch of that predicate. The predicate of cell c is "$(a < 5) and (not (b > 3))$". Since this predicate is an *and* type predicate, we first calculate the values of the two sub branch functions based on the rules given in Tables 3.1 and Table 3.2: $v_1 = f(a < 5, true) = 5 - a = -2$, $v_1 = f(not\ (b > 3), true) = -f(b > 3, true) = 3 - b = -12$. According to the rule in Table 3.2 and the values of v_1 and v_2, we obtain the branch function $F = f(a > 5, true) + f(not(b < 3), true) = 5 - a + (-(b - 3))$. The value of this branch function associated with the current input value is $(-2) + (-12) = -14$, which indicates the goal is not achieved and the distance between the current input values and the solution is -14. Assume cell b's value is changed to 0 and cell a is kept at 7. To obtain the branch function, we recalculate the values of the two sub branch functions based on the current input values. Now the value of the second sub branch function is +3 and the value of the first sub branch function is still -2, which indicates the current branch function $F = min\{f(a < 5, true), f(not(b > 3))\} = 5 - a$, and the value is -2. We can see that we still have not achieved the goal (since the branch function is still negative) but we are closer to the goal (the value changed from



```
if ( (a < 5) and (not  (b > 3)))
then a
else b
```

FIGURE 3.6: A simple spreadsheet

-14 to -2). Next, assume we change cell a's value to 4 and cell b's value remains 0. Under these input values, the values of both sub branch functions are positive. So the value of the branch function under these input values is $F = min\{f(a < 5, true), f(not(b > 3))\} = 5 - a = +1$. The positive result indicates that our goal is achieved.

The rules for creating branch functions for predicates could also apply to predicates that contain multiple boolean operators. For example, if the form of the predicate is E_1 *and* E_2 *and* E_3, we could first transform this predicate to the form (E_1 *and* E_2) *and* E_3, then apply the rules to it as a simple boolean type predicate. These rules are suitable for predicates in imperative programs too. However, our method for creating branch functions has some limitations.

For example, we have not yet considered predicates that contain only a single boolean type variable or a function that returns a boolean type value, such as " $IsInteger(i)$" where $IsInteger$ is a function that returns a boolean value. These limits must be addressed in future work.

3.3.4 Task3: searching for the solution

We now describe how we search for the solution of a subgoal under the guidance of the branch function. The whole search procedure is performed by comparing the successive values of the branch function. If the branch function is negative and closer to 0, we consider that we are closer to the solution. Similar to the search in Korel's goal-oriented method [15], the search procedure in our goal-oriented technique varies the input cells in turn, aiming to increase the value of the branch function. In order to speed up the search procedure, we consider only those input cells that could affect the value of the branch function. We determine these by doing backward data dependence analysis on data provided by the WYSIWYT methodology. We alternatively perform a *one-dimensional search* on each cell in the set of relevant input cells until the subgoal is achieved or no progress toward the subgoal can be made searching any input cells. In the latter case, it means that we have failed to exercise the desired branch.

A one-dimensional search on an input cell starts with an "exploratory search", then turns into a "pattern search". The exploratory search moves the value of the given input cell by a small step. Through this small probe, it determines whether changing the given cell could make any progress toward the subgoal. If it could not, then the one-dimensional search terminates. If it could, this suggests a search direction for the pattern search. Assuming the initial probe shows that changing the given input cell can make progress, the pattern search continues searching on that cell in the direction suggested by the initial search using a larger move, monitoring the branch function and the possible constraint for violation. If the branch function is improved, the pattern search continues with a doubled move step. This procedure continues until no progress is made or a constraint is violated. Then the move step is reduced as necessary.

We illustrated how we create branch functions when the input values change through an example in the previous subsection. Now we will illustrate how we search for input values under the guidance of the branch function using the same example. Assume our subgoal is still to exercise the true branch of the predicate in cell c. The initial values in cell a and cell b are 7 and 15, respectively. The value of the branch function under the current execution is -14. We begin by obtaining the input cells which may affect the predicate: in this case, cell a and cell b. Assume we begin with cell b and keep cell a fixed. An exploratory search on cell b decreases cell b's value by 1 and improves the branch function by 1 (from -14 to -13). This suggests that searching in the decreasing direction might improve the branch function. Thus, we perform three continuously successful searches which decrease cell b by 2, 4, and 8 respectively, with cell b's value changed to 0 and the value of the branch function changed to -2. Since all three previous searches have been successful, we decrease cell b by 16 (to a value of -16). Since the second subpredicate is already satisfied when b's value is 0 but the first subpredicate is still not satisfied, any improvement on the sub branch function does not improve the whole branch function. Since the branch function is not improved, we reset cell b's value to 0 and try to decrease cell b with a smaller step. Finally the search on cell b terminates since no further move on it improves the branch function. Now we turn our attention to cell a and keep

b's value at 0. In a similar way, the exploratory search detects that decreasing cell a's value by 1 improves the branch function to -1. Decreasing by 2, cell a's value is changed to 4, and the branch function is changed to +1. A solution has been found and the desired branch is exercised.

The search strategy we used here is directly searching on input cells. Its effectiveness is affected by the depth of a du pair (the levels of cell reference between the input cells and a du pair). When a du pair's depth is great, the small progress made by exploratory search may be absorbed by the intermediate cells and is not reflected in the value of the branch function of that du pair. In this case, the exploratory search could not suggest a searching direction for the pattern search. Therefore, it is possible for the goal-oriented technique to fail to find a test case to exercise that du pair.

3.4 ATCG from the user's view

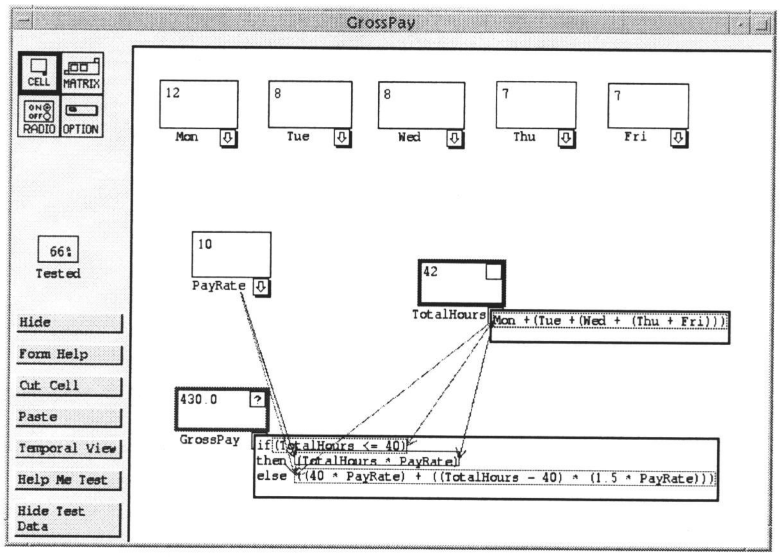
We prototyed our ATCG methodology for spreadsheets in Forms/3. The visual effect of our initial implementation could be illustrated by the example given in Figure 2.2. Suppose an end user is partially through testing this spreadsheet. The testedness indicator located on the left side of the control panel shows that 66% of the du pairs in the spreadsheet have been tested. If at that time the end user wants help finding another test case that increases testedness, she could ask for help from the underlying ATCG technique by clicking the "HelpMeTest" button in one of three ways:

1. She could ask for help to generate test case for this spreadsheet by directly clicking the "HelpMeTest" button. The ATCG technique will attempt to generate a test case that could increase the testedness of that spreadsheet.

2. The colored border of cell "GrossPay" may alert the end user that cell "GrossPay" is not completely tested, and she may wish to increase the testedness of cell "GrossPay". She could ask for a test case for cell "GrossPay" by selecting that cell, and then clicking on the "HelpMeTest" button. In this case, our underlying ATCG technique will work on that selected cell and attempt to generate a test case that will increase the testedness of the selected cell.

3. Through the colored arrows, the end user may know which particular interactions (definition-use pairs) are untested. She could ask for help from our ATCG methodology by simply selecting the arrow associated with that untested du pair and clicking the "HelpMeTest" button. Our underlying ATCG technique will find out which du pair the user is interested in, and generate a test case that could allow that du pair to be tested.

During the generation procedure, input cells' values are kept unchanged to the end users' view, until the underlying ATCG technique generates a test case successfully. Then the system automatically updates the input cells' values, showing the test case and its effects on the screen. There will be a question mark in some output cells to prompt the end user to validate the results associated with the generated test case. If the end user prefers to try a different test case, our ATCG methodology will help in the same way but will generate a new test case. Figure 3.7 illustrates the visual feedback given by the ATCG methodology after it generates a test case as the user requested. The check box of cell *GrossPay* changes to contain a question mark, indicating that some du pairs that have not been tested before are exercised under the current inputs, and cell *GrossPay* is one sink cell of those du pairs.



```
Mon + (Tue + (Wed + (Thu + Fri)))
```

```
if ((TotalHours <= 40)
then (TotalHours * PayRate)
else ((40 * PayRate) + ((TotalHours - 40) * (1.5 * PayRate)))
```

FIGURE 3.7: Spreadsheet GrossPay. Inputs show a new test case automatically generated after a user has requested help.

3.5 Evaluating the methodology

There are many questions to be empirically investigated to determine whether our methodology works, including the question "can people use it?" But user studies are complex, and before we consider that question, there are a number of others to consider. For example, if we cannot generate inputs effectively, there is no point in conducting any user study. Thus, as our first question we investigate whether our ATCG techniques can find test cases to execute most dupairs in spreadsheets.

Since our ATCG methodology could assist end users in generating test cases at three levels, the whole spreadsheet level, the cell level and the du pair level, our initial thought was to conduct a family of experiments which investigate our methodology at those three levels. The experiment at the whole spreadsheet level is necessary since an end user may solely depend on our ATCG methodology to generate test cases to test the whole spreadsheet. However, empirical studies of our WYSIWYT testing methodology show that it is not too difficult for testers to find some test cases by themselves initially [27]. Eventually, however, users find it difficult to find test cases to cover untested du pairs. This suggests that the request for testing part of the spreadsheet is often important to meet too. It seems that experiments at the cell level and at the du pair level could both examine this. Since generation at the du pair level addresses the generation process for an individual du pair, whereas generation at the cell level is similar to the generation at the whole spreadsheet level in generating test cases for a list of untested du pairs, we choose to perform experiments at the du-pair level and at the spreadsheet level in this study. Future studies can investigate the methodology at the cell level.

Chapter 4

EMPIRICAL STUDIES ONE: SPREADSHEET LEVEL

To empirically examine the effects of employing ATCG techniques at different levels in spreadsheet languages, we conducted a family of experiments. Our experiments focus on the random and the goal-oriented techniques described in Chapter 3. Since the main goal of ATCG is to help testers generate test cases for a whole spreadsheet, our first empirical study is designed to evaluate the two test case generation techniques when they are applied to whole spreadsheets.

An additional consideration is whether the availability of *explicit range information* — stated bounds on the expected values of input cells — affects test generation techniques. Sometimes end users may have knowledge of the expected range of values that might be given to input cells, and using this explicit range information, ATCG techniques may be more effective and efficient. In some cases, however, especially with end users who have no testing background, it may be difficult for them to determine what these input ranges should be. In such cases, users may depend solely on the test case generator without providing any hints. Thus, our ATCG techniques are designed to work both with and without explicit range information. To empirically examine effects related to range information, we implemented two experiments at the whole spreadsheet level. This chapter presents these experiments.

We begin with a discussion of issues common to both experiments in Section 4.1 through Section 4.5. Section 4.6 presents the design and results of an experiment without explicit range information. In Section 4.7, we describe the design and the results of experiment with explicit range information. Then, we discuss the threats to validity for these experiments in Section 4.8. Finally, a discussion is given in Section 4.9.

4.1 Research questions

At the whole spreadsheet level, we are interested in the following questions:

RQ1: Can we automatically generate test cases that execute a large proportion of the feasible du pairs in a spreadsheet at the whole spreadsheet level, either with or without range information?

RQ2: How do the two test case generation techniques we consider compare to each other in terms of effectiveness and efficiency at the whole spreadsheet level, with or without range information?

4.2 Measures

To measure a test case generation technique's effectiveness at the whole spreadsheet level, we measure the quality of the test cases it generates. Since our underlying testing system uses du-adequacy as a testing criterion, we use the proportion of feasible du pairs executed by the generated test cases as our measurement of effectiveness. We gather this metric incrementally over the course of testing: we automatically record the new cumulative testedness whenever a new test case is generated.

Since our test case generation methodology is designed to help end users automatically generate test cases, efficiency is also an important issue. To measure a test case generation technique's efficiency at the whole spreadsheet level, we measure the speed of test case generation: the clock time needed to generate test cases sufficient to achieve various levels of testedness.

4.3 Subjects

In this study we used eight relatively small but nontrivial spreadsheets as subjects. Most of these spreadsheets had been used previously in another study of the WYSIWYT methodology [24, 27]. Table 4.1 provides some data about these subjects. These spreadsheets perform a wide variety of tasks: Digits is a number to digits splitter, Grades translates quiz scores into letter grades, FitMachine and MicroGen are two simulations, NetPay calculates an employee's income after deductions, PurchaseBudget determines whether a proposed purchase is within a budget, Solution is a quadratic equation solver, and NewClock is a graphical desktop clock.

Since our initial implementation of the test case generation techniques described in Chapter 3 handles only integer type inputs, all input cells in these subject spreadsheets are of integer type. Since commercial spreadsheets contain infeasible du pairs [27], all subject spreadsheets in our experiments also contain infeasible du pairs. We determined all the infeasible du pairs through careful inspection.

spreadsheets	No. of cells	No. of du pairs	No. of feasible du pairs	No. of expressions	No. of predicates
Digits	7	89	61	35	14
Grades	13	81	78	42	12
MicroGen	6	31	28	16	5
NetPay	9	24	20	21	6
PurchaseBudget	25	56	50	53	10
Solution	6	28	26	18	6
NewClock	14	57	49	39	10
FitMachine	9	121	101	33	12

TABLE 4.1: Data about experimental subjects

4.4 Experiment environment

We prototyped the two ATCG techniques presented in Chapter 3, the random technique and the goal-oriented technique, in Forms/3. We chose Forms/3 because we have access to its implementation, and therefore, we could easily implement and experiment with ATCG techniques within its environment.

4.5 Method

Examining our research questions requires us to apply our ATCG techniques to our subject spreadsheets and collect our measures of interest. However, no end users are involved in this study; rather, our experimentation requires us to simulate end users by applying our ATCG techniques multiple times to multiple spreadsheets, in a controlled fashion. This requires automation; thus we use automated scripts in our experiments. These scripts repeatedly invoke our test case generation techniques and gather measurements. The use of these scripts raises several issues. We describe these issues and how our scripts address those issues here.

4.5.1 Automatic validation

Validation plays an important role in the WYSIWYT methodology. The whole testing procedure under WYSIWYT is divided into two steps: first, finding a test case that executes one or more untested du pairs in the spreadsheet; second, validating output cells as prompted to mark executed du pairs as "tested". Since we are interested only in the test case generation procedure and do not have users performing validation, our scripts automatically validate all validatable output cells whose validation would cause some du pair to be considered exercised.

More precisely, after any new test case is generated, our scripts check all the cells affected by the changed input cells. Those cells that are sinks of du pairs not previously validated, but that are now exercised by the given test case, are considered to be *validatable cells.* Our scripts automatically validate all such validatable cells.

4.5.2 Feasible and infeasible du pairs

For the purpose of measuring effectiveness, we consider only coverage of feasible du pairs; this lets us make fair comparisons between subject spreadsheets. We can do this since we already know the infeasible du pairs for all subject spreadsheets through analysis. However, in practice, our ATCG techniques would be applied to spreadsheets containing both feasible and infeasible du pairs. So we keep the infeasible du pairs in our subject spreadsheets when we apply the two ATCG techniques. In this way, we can also initially investigate our ATCG techniques' effectiveness at identifying infeasible du pairs, through comparing the du pairs remaining unexercised with the known infeasible du pairs.

4.5.3 Time limit

When used by an end-user, ATCG techniques generate only one test case at a time. In our study, to force our ATCG techniques to continuously generate test cases, in addition to applying automatic validation, our scripts continuously apply the ATCG techniques to the subject spreadsheet after each auto-validation. To address our first research question, our scripts must provide enough time for our ATCG techniques to generate test cases. Obviously the techniques would stop if 100% du-adequacy were achieved; however, since each subject spreadsheet contains infeasible du pairs, and our generators are not informed as to which du pairs are executable, this condition will never occur. Moreover, even for spreadsheets that contain no infeasible du pairs, we do not know whether the test case generation techniques we consider could generate test cases for all feasible du pairs. Thus, we use a timer with a time limit sufficient to make our ATCG techniques reach a likely limit in their generation ability, as well as stop them in case only infeasible du pairs remain.

In order to investigate what time limit to use in our experiments, we performed several trial runs with very long time limits per script. We found from the results that for all subject spreadsheets, no additional test cases were found after the scripts had run for 1200 seconds. Of course, in general, it is possible that with additional time, additional test cases would be discovered. However, our runs suggest that this is unlikely. In addition, since our ATCG methodology is designed for the end users, users may prefer quick feedback to a long generation process. Moreover, we attempt to balance the need for the longer time limit per script with the overall time needed for all the empirical studies. Thus we used 1200 seconds as the time limit in our scripts.

4.5.4 Initial values

Another consideration that might affect the effectiveness and efficiency of ATCG techniques is the initial values present in spreadsheet cells when testing begins. The random test case generation technique randomly generates input values until it finds a "useful" one, while the goal-oriented technique starts from the current value the input cells have, and searches the input space under the guidance of a branch function until it finds a solution. Thus, the random technique is independent of initial values whereas they could affect the goal-oriented technique. It follows that our empirical results on the goal-oriented technique (and thus our comparisons of the two techniques) may vary under different initial values. To control for this possibility, we run our experiments 35 times on each spreadsheet, using different input values for each run. To facilitate our comparison of the ATCG techniques, in each run, both techniques start from the same set of initial values.

4.5.5 Range information

Input range information is another important consideration for ATCG techniques. The random technique requires a range within which to randomly select an input value, and the goal-oriented technique needs to know the edge of its search space. Since all input cells in our subject spreadsheets are of integer type, here we consider only the provision of ranges for integer type inputs. We discuss two ways to meet this requirement here.

One possible scenario is that the user may depend solely on the test case generator to generate test cases without providing any hints about input ranges. In that case, with no explicit range information available, the ATCG techniques will consider all possible cell values within the default range of the data type. Our first experiment at the whole spreadsheet level is designed to model this scenario. On our system, the default range used was -536870912 to +536870911. We determined that this default range is large enough to provide inputs that can execute every feasible du pair in each of our subject spreadsheets.

A second possible scenario is that via a user's help or a range information analysis tool, the ATCG techniques could obtain more precise knowledge of range information. With explicit ranges, both techniques will limit their search space to the specified ranges and generate test cases exactly within these ranges. This might improve test case generation. To investigate this possibility and investigate research questions concerned with explicit ranges, we employed another experiment. In this experiment, the two techniques generate test cases based on ranges provided per input cell. No tool or user's help are available in Forms/3 to obtain input ranges at present. Thus, to obtain range information for all input cells in our subject spreadsheets, we carefully examined the spreadsheets, considering their specifications and their formulas. Then we created an original range for each input cell that seemed appropriate based on this examination.

The range information we initially created specified only input values expected in normal cases. This might create a problem. To achieve 100% test adequacy, test cases should not only exercise expected inputs, but also exercise inputs that may lead to error cases in the spreadsheet. Users might be particularly interested in the test cases that detect these error cases to uncover potential faults. In practice, users might expand ranges to include error values; alternatively, ATCG tools might expand ranges. To model this, we chose to include input values outside of expected ranges by expanding our initial ranges by 25% in both directions. These expanded ranges were the ones we used in this experiment.

4.5.6 Experimental procedure

As Figure 4.1 shows, for each spreadsheet S, for each of the 35 runs, we did the following. First we loaded S into Forms/3 (line 4), and then we randomly selected a set of input values within the target ranges (explicit or default), applied them to the input cells in S, and then we saved S (lines 5-6). Next (lines 7-11) for each technique (random and goal-oriented) we loaded S again with the same initial values[1] and repeatedly applied the technique to S until time out.

[1] In this algorithm, to reduce the possibility that our timing measurements would be inappropriately influenced by Forms/3 caching, we completely quit Forms/3 and reenter Forms/3 before apply each ATCG technique to a subject spreadsheet.

```
Algorithm FormsATCG
Input: Set of subject spreadsheets $SS$ , set of ATCG techniques $TG$
Output: Set of result files $RF[35][8][2]$
1.begin
2.    for each spreadsheet $S$ in $SS$
3.       for (run = 1; run $\leq$ 35 ;run ++)
4.          invoke Forms/3 and load $S$
5.          randomly initialize input values within ranges and apply them to input cells in $S$
6.          save spreadsheet $S$ and exit Forms/3
7.          for each ATCG techniques $T$ in $TG$
8.             invoke Forms/3 and load $S$ /* with input values saved in step 6 */
9.             $R$= ApplyATCGtoForm($S$,$T$)
10.            write Record $R$ to result file $RF[run][S][T]$, then exit Forms/3
11.         endfor
12.      endfor
13.   endfor
14. end
Procedure ApplyATCGtoForms $S,T$
Input: Subject spreadsheet $S$, ATCG technique $T$
Output: Record of testedness and time $RL$
15.begin
16.   while not time out
17.      apply ATCG technique $T$ on spreadsheet $S$
18.      if new test case $TC$is generated
19.         for each validateable cell $C$ arisen by test case TC
20.            Validate($C$)
21.            record testedness and time in $RL$
22.         endfor
23.      endif
24.   endwhile
25.   return $RL$
26. end
```

FIGURE 4.1: Algorithm for experiments at the whole spreadsheet level

4.6 Experiment 1A: with no explicit range information

Our first experiment evaluated automatic test case generation techniques on eight subject spreadsheets without explicit range information.

4.6.1 Experiment design

The two independent variables manipulated in this experiment are:

- The eight subject spreadsheets
- The test case generation techniques: *Random* and *Goal-oriented*

We measured 2 dependent variables:

- testedness
- Average time needed to reach successive levels of testedness

This experiment was run using an 8×2 factorial design with 35 different initial input configurations per spreadsheet. For each subject spreadsheet F, we applied each of our two test case generation techniques starting from 35 sets of initial inputs. On each run, we measured the times at which untested du pairs were exercised. These measurements provided the values for our dependent variables. These runs yielded 560 sets of testedness and time values for our analysis.

4.6.2 Data and analysis

Fig 4.2 depicts the mean cumulative testedness achieved over the 1200 seconds by the two test case generators in 35 runs. Each plot depicts results for one subject spreadsheet. In the plots, the two lines represent the mean cumulative testedness achieved over time across the 35 runs. The darker line represents the random technique and the lighter line represents the goal-oriented technique.

As we can see from the plots, the goal-oriented technique achieved coverage gradually. The generated test cases eventually executed more than two-thirds of all feasible du pairs for all subject spreadsheets: final testedness ranged from 0.68 to 1.0. On three of the eight spreadsheets (MicroGen, NetPay and Solution), the goal-oriented technique eventually achieved 100% du coverage; on another three spreadsheets (PurchaseBudget, NewClock, and Fit-Machine), although the goal-oriented technique did not exercise all the du pairs, it achieved greater than 90% du coverage eventually; on Grades and Digits, the goal-oriented technique achieved 82% and 68% du coverage, respectively.

The random generation technique typically achieved coverage rapidly at the beginning, then achieved no additional success over time. The final testedness reached by the random technique varied from 0.26 to 0.96. For PurchaseBudget, the random technique reached almost 100% du coverage; for Grades and MicroGen, it covered about 70% of the feasible du pairs; for the other four spreadsheets, it covered less than 60% of the du pairs. Digits was especially troublesome: only 26% of its feasible du pairs were exercised by the test cases generated by the random technique.

Comparing the two techniques, we can see that, for most spreadsheets, the testedness eventually achieved by the goal-oriented technique is noticeably higher than the testedness achieved by the random technique. In two exceptional cases, Grades and PurchaseBudget, the goal-oriented technique achieved slightly less testedness than the random technique did. On the other hand, the goal-oriented technique did not always occupy the leading position over

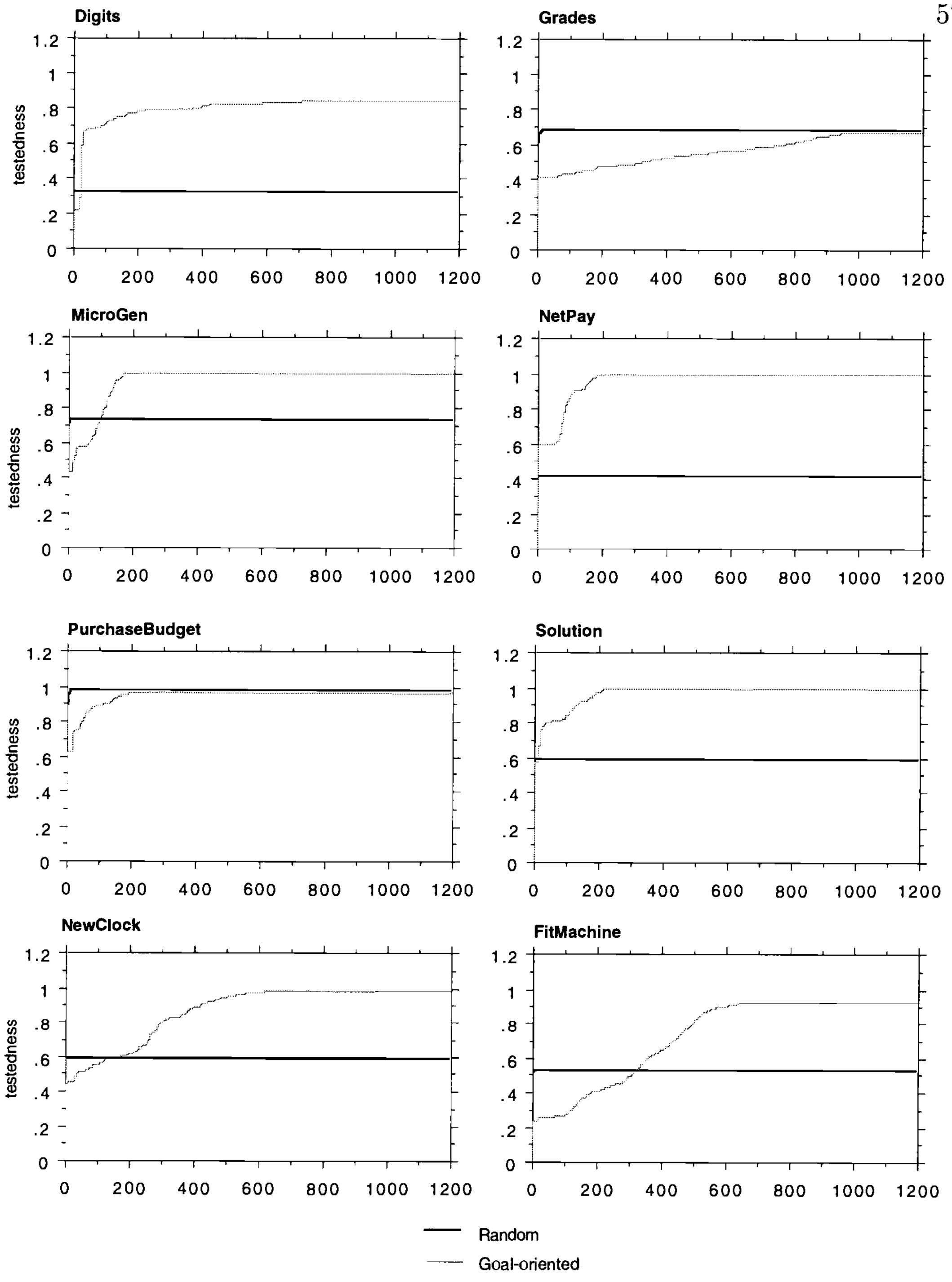


FIGURE 4.2: Test case generation efficiency and effectiveness at the whole spreadsheet level with no explicit range information provided. Graphs show the average cumulative testedness (ranging from 0.0 to 1.0) over 1200 seconds on eight subjects in Experiment 1A.

the whole generation procedure. On six spreadsheets, the random technique achieved coverage faster than the goal-oriented technique initially. However, the random technique did not progress further after a brief period, whereas the goal-oriented technique, although slower initially, continued to make progress. For NetPay and Solution, it seems that the random technique lost its ability to make progress so soon that the goal-oriented technique was always better.

Observing the mean cumulative testedness achieved in Figure 4.2, we can see that the speeds of generation are different for the two ATCG techniques. In that figure, the two lines representing the two techniques usually cross at some time. In some spreadsheets, the thin line (representing the goal-oriented technique) achieved lower testedness initially than the thick line (representing the random technique). Later the thin line grows faster than the thick line, achieving the same mean cumulative testedness at some time and then eventually reaching higher testedness than the thick line.

Figure 4.2 only depicts the mean cumulative testedness achieved by the two techniques over the 35 runs. We are also interested in the distribution of the testedness values achieved by the two techniques at a particular time over the 35 runs, and this requires further analysis. However, comparing the testedness achieved by the two techniques at every second over the total 1200 seconds is not necessary. Instead, we choose to compare the results over the 35 runs at two times. Time T_1 is a time before the time when the two techniques achieved the same testedness (if they did not meet at any time, then the 10 second mark is selected); time T_2 is a time after testedness achieved by the two techniques has reached its final level. Since for different spreadsheets, the times at which the two techniques achieved the same mean cumulative testedness are different, the times T_1 and T_2 used in our analyses differ across our spreadsheets.

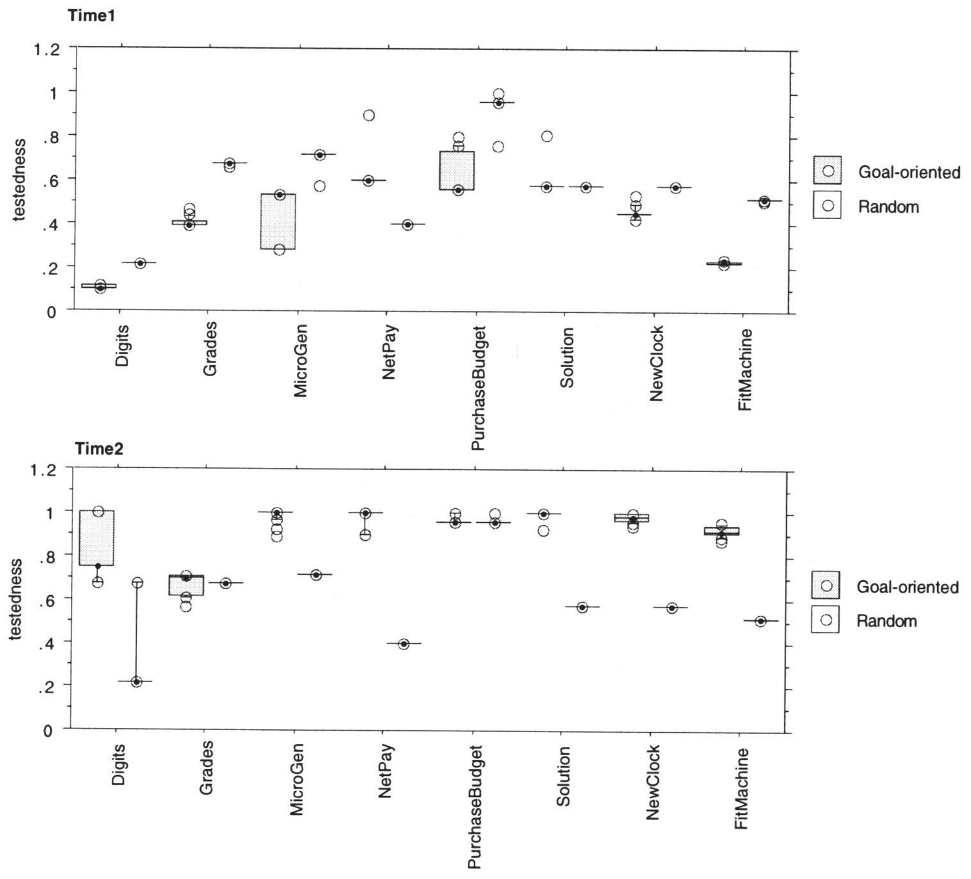


FIGURE 4.3: Test case generation efficiency at the whole spreadsheet level with no explicit range information provided. Boxplots show the distribution of testedness (ranging from 0.0 to 1.0) at two times. For each spreadsheet, two box plots are given: the left plot depicting data for the goal-oriented technique, the right plot depicting data for the random technique.

Figure 4.3 shows box plots of the cumulative testedness achieved at times T_1 and time T_2 by the two ATCG techniques on the eight spreadsheets. A box plot is a graphical representation of a data set that visually depicts the distribution and the direction of skewness. In each box plot, the box indicates the range in which the middle half of the data falls (interquartile range). The line with a bold dot denotes the median. The whiskers above and/or below boxes indicate ranges over which the lower 25% and upper 25% of the data falls, respectively. All other data points that fall within a distance greater than 1.5 times the interquartile range are considered outliers, represented by small circles. As the box plots at time T_1 show, on six of the eight spreadsheets, the random technique achieved greater testedness than the goal-oriented technique did at time T_1. The goal-oriented technique only achieved greater testedness at time T_1 on NetPay. On Solution, the techniques achieved the same testedness. At time T_2, the goal-oriented technique achieved noticeably greater testedness on six spreadsheets. On Grades and PurchaseBudget, the differences are not so obvious. To formally assess the differences in testedness achieved by the two techniques over the 35 runs at times T_1 and T_2, we performed a group of paired t-tests, in which if the p-value is less than or equal to 0.05 the mean difference is considered significant. The results displayed in Table 4.2 confirm our box plot observations. In addition, they show that at time T_1, on all subjects, the testedness achieved by the two techniques were widely different; at time T_2, results were widely different on four subjects.

4.7 Experiment 1B: with explicit range information

To address our research questions concerning the use of range information, we performed another experiment using such information.

4.7.1 Experiment design

The experiment was identical to the first experiment, using the same subject spreadsheets and design, except that explicit ranges were used.

	Time 1			Time 2		
spreadsheets	Mean Diff.	t-Value	P-Value	Mean Diff.	t-Value	P-Value
Digits	-.109	-78.483	<.0001	.538	13.658	<.0001
Grades	-.266	-78.653	<.0001	-.001	-.101	.9200
MicroGen	-.282	-13.543	<.0001	.279	72.989	<.0001
NetPay	.218	17.712	<.0001	.586	97.600	<.0001
PurchaseBudget	-.320	-17.649	<.0001	.006	1.966	.0576
Solution	.007	1.000	.3244	.419	136.779	<.0001
NewClock	-.119	-30.672	<.0001	.409	144.225	<.0001
FitMachine	-.284	-236.863	<.0001	.406	104.650	<.0001

TABLE 4.2: Paired t-test for Experiment 1A

4.7.2 Data and analysis

Similar to Figure 4.2, Figure 4.4 shows the average cumulative testedness achieved over time by the generated test cases when providing explicit range information for the input cells, for the two techniques. Since these two techniques made no additional progress in the later 400 seconds of the 1200 seconds time limit, we display the generation effectiveness and efficiency over the first 800 seconds.

Examining Figure 4.4, the graphics show that the use of explicit ranges produced improvements in final testedness in all cases in which improvements were possible. More than half of the subjects were almost fully tested by the random technique, and the goal-oriented technique reached greater than 85% testedness on all spreadsheets. Solution was the only spreadsheet on which the random technique did not achieve within 15% of the goal-oriented technique: the random technique exercised only 63% of its du pairs, whereas the goal-oriented

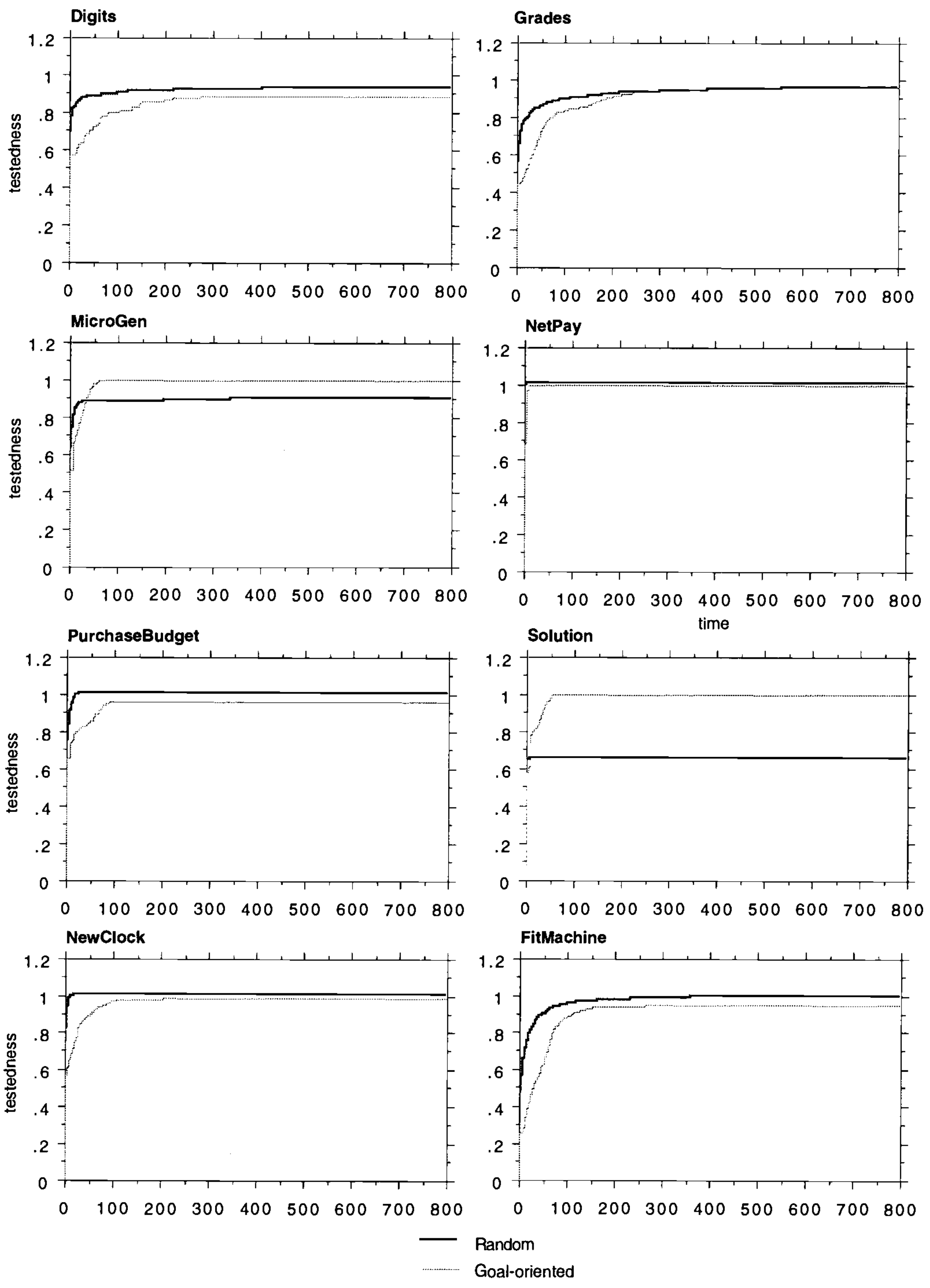


FIGURE 4.4: Test case generation efficiency and effectiveness at the whole spreadsheet level with explicit range information provided. Graphs show the average cumulative testedness (ranging from 0.0 to 1.0) over 1200 seconds on eight subjects in Experiment 1B.

technique achieved noticeably larger testedness (100%). For the other seven subjects, the final testednesses achieved were not widely different. In addition, considering overall results, both techniques made progress on the rate of generation at the beginning: the goal-oriented technique is faster than it was without range information. However, the random technique still achieved coverage a little faster initially than the goal-oriented technique on most subjects.

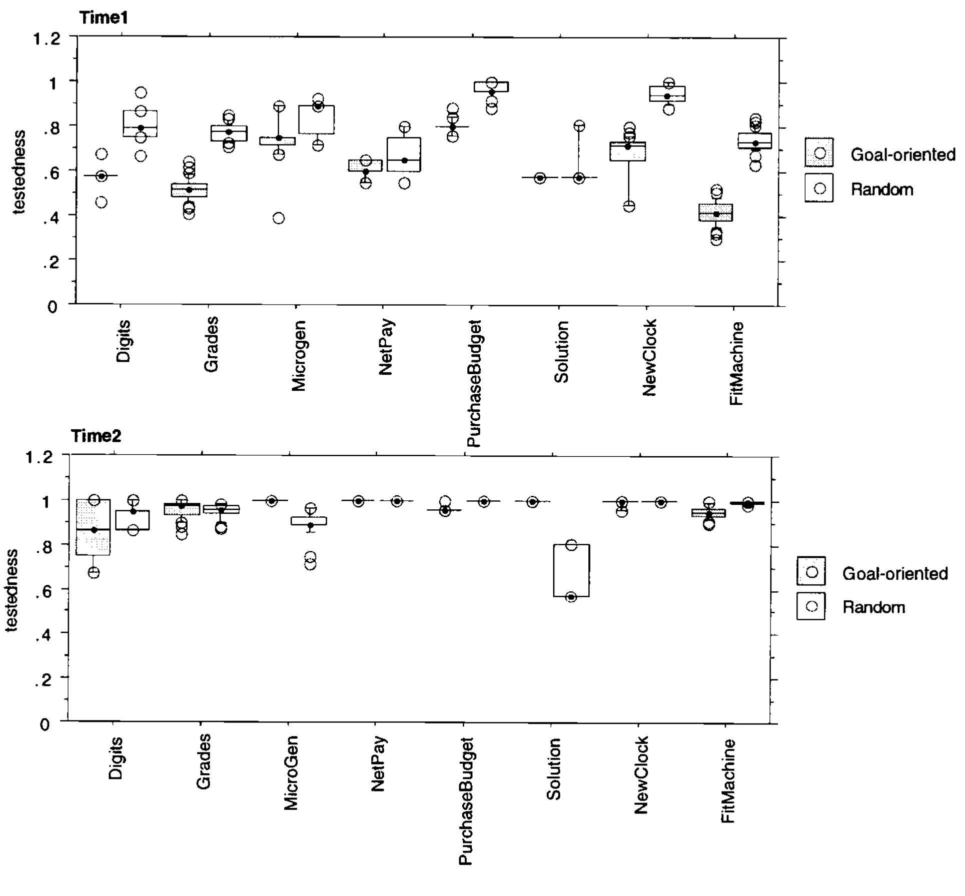


FIGURE 4.5: Test case generation efficiency at the whole spreadsheet level with explicit range information provided. Boxplots show the distribution of testedness (ranging from 0.0 to 1.0) at two times. For each program, two box plots are given: the left plot depicts data for the goal-oriented technique, the right plot depicts data for the random technique.

As in Experiment 1A, we compared the results of the two techniques at two different times over the 35 runs for the eight spreadsheets. Since the times at which various levels of testedness are achieved by the two techniques changes after explicit range information is provided, we use different times T_1 and T_2 than those used in Experiment 1A. Times T_1 and T_2 are selected in the same way as in Experiment 1A. The resulting box plots are displayed in Figure 4.5. These box plots illustrate that, at time T_1, the goal-oriented technique achieved noticeably lower testedness than the random technique did on six spreadsheets, however, on NetPay and Solution, the techniques are not widely different. At time T_2, however, the results are distinguishable only on three spreadsheets: MicroGen, Solution, and FitMachine. Our observation on these box plots are confirmed by the results of paired *t*-tests shown in Table 4.3.

	Time 1			Time 2		
spreadsheets	Mean Diff.	t-Value	P-Value	Mean Diff.	t-Value	P-Value
Digits	-.229	-16.733	<.0001	.035	-1.519	.1379
Grades	-.255	-20.852	<.0001	.007	.660	.5140
MicroGen	-.103	-4.293	.0001	.102	9.938	<.0001
NetPay	-.079	-4.639	<.0001	0.000	0.0000	1.0000
PurchaseBudget	-.171	-24.258	<.0001	-.039	-34.000	<.0001
Solution	-.046	-2.915	.0062	.357	19.974	<.0001
NewClock	-.279	-14.285	<.0001	-.006	-2.533	.0161
Fit-machine	-.320	-27.795	<.0001	-.043	-8.978	<.0001

TABLE 4.3: Paired t-test for Experiment 1B

4.8 Threats to validity

The potential threats to validity for our first studies are as follows.

4.8.1 Threats to external validity

Threats to external validity are conditions that limit the ability to generalize the results of the studies to a larger population of subjects. We considered several such threats:

- Subject program representativeness. The subject spreadsheets used in these experiments are of small and medium size and the input cells are of integer type. Commercial complex spreadsheets with different characteristics may be subject to difference cost-effectiveness trade-offs. These threats can be addressed only through additional studies using other spreadsheets. As a first step in this direction, later in this thesis, we describe results obtained with two larger spreadsheets.

- Validation process representativeness. In our experiment, we use a script to auto-validate all validateable output cells. In reality, a user may validate some of these cells or none of them. Further studies involving end users are necessary to address this threat.

- Range information representativeness. In our experiment at the whole spreadsheet level with explicit range information, the ranges we created may not represent the ranges that would be specified in practice by end users. We attempt to address these threats by carefully examining specifications and formulas of subject spreadsheets, and including values outside of expected ranges.

- Initial value representativeness. The initial values used in our scripts may not be representative of the ones used by end users. These threats are initially addressed through randomly generating values within provided ranges (or default ranges). This threat could be reduced only by further studies with end users.

4.8.2 Threats to internal validity

Threats to internal validity are factors that can affect the dependent variables without the researcher's knowledge. We considered the following threats:

- The differences among subject spreadsheets may affect results. Some spreadsheets use many conditional expressions that contain the "=" operator, these are difficult for the random technique to address within a large range. Some spreadsheets have several levels of dependency among cells; these will cause the goal-oriented technique to require more effort to satisfy breakpoints to reach a definition node or a use node. To limit these threats, we experimented using a range of spreadsheets that perform a wide variety of tasks.

- Initial input cells' values can affect the success of the goal-oriented technique. If the goal-oriented technique starts from initial values that are far away from the solution, it will spend more time to search than if it starts with values that are close to the solution. We address this threat by starting to run our ATCG techniques from 35 sets of different initial values for each subject spreadsheet.

- Another source of threats involves collecting timings of our implementations, and comparing two implementations. To control these threats, we: 1) ran our experiments on isolated machines to avoid outside influences 2) fully exited Forms/3 and lisp between executions to avoid caching 3) were careful in our implementations to keep as much code in common between the two tools as possible, varying only the code unique to each technique.

4.8.3 Threats to construct validity

Threats to construct validity occur when measurements do not adequately capture the concepts they are supposed to measure. Two kinds of construct threats are considered here:

- The degree of testedness achieved by generated test cases is not the only possible measure of the effectiveness of ATCG techniques. If two groups of test cases achieve the same testedness, the one with smaller size may be preferable. In addition, a test case's fault detection ability is another important measure that we did not consider here. Moreover, two test cases may execute the same number of du pairs, but one may create output values that are easier for users to validate than another. Additional studies are necessary to address these threats.

- Efficiency is often measured as the effort required to obtain a certain results. We use the wall clock time required to generate test cases sufficient to achieve various levels of testedness to measure the generation efficiency in our study. However, time is not the only possible measure of effort. Other measures, such as the number of failed test inputs tried before the ATCG technique finds a test case, or the memory required to store the data required in calculation, could also measure effort. However, since our final goal in using ATCG is to help end users, the feedback time seems to be the most worthwhile measure to address here. Further studies could investigate other measures.

4.9 Discussion

Overall, our results in this study indicate that the goal-oriented ATCG technique could execute a large proportion of the feasible du pairs in a spreadsheet, either with or without explicit range information. Overall the random ATCG technique, although exercising almost 100% of the feasible du pairs in one spreadsheet, exercised fewer than 75% of the feasible du pairs in the other seven spreadsheets when no explicit range was provided. However, when explicit range information is provided, the random technique exercised about 90% of the feasible du pairs in general.

Further examination of our results illustrates that there are many factors that may account for differences in the effectiveness of the two ATCG techniques. One such factor is explicit range. Table 4.4 shows the effects on testedness, for the two techniques, of providing explicit range information. The table lists the increase in testedness when ranges are used. The data in this table suggests that the influence of explicit ranges is noticeably greater on the random technique than on the goal-oriented technique. Considering the way in which the random technique generates test cases, it is not hard to understand why the explicit ranges are particularly "useful" for its generation: it randomly selects values from the provided ranges, and the likelihood it can select a useful test case is enhanced when the provided ranges are smaller. In contrast, since

spreadsheet	Goal-oriented	Random
Digits	5.2%	202.5%
Grades	43%	42%
MicroGen	0%	26%
NetPay	0%	150%
PurchaseBudget	-0.7%	3.9%
Solution	0%	11.4%
NewClock	1.4%	75%
FitMachine	3.2%	92.9%

TABLE 4.4: Percentage increase in testedness achieved by the two techniques with explicit ranges, as compared to testedness achieved without explicit ranges.

the goal-oriented technique searches for test cases under the guidance of the branch function, its likelihood of finding a useful test case is not as noticeably enhanced by explicit ranges.

Another factor that may account for the effectiveness differences is the types of formulas occurring in the subject spreadsheets. Examining the results and the subject spreadsheets, we found that the types of expressions used in the spreadsheets can greatly influence both techniques, but especially so the random technique. If the proportion of predicate expressions over all expressions is higher in a spreadsheet, then it is often more difficult for the random technique to exercise a large proportion of the feasible du pairs in that spreadsheet. For instance, the proportion of predicates over all expressions is 33.33% in Solution, ranking the highest over the eight subject spreadsheets; while it is only 18.87% in PurchaseBudget, ranking the lowest (see Table 4.1). Examining the results in Figure 4.2, we see that the random technique achieved distinctly smaller testedness than the goal-oriented technique did on Solution; whereas there is no difference between the techniques on the final testedness achieved on PurchaseBudget. This also explains the differences in the effectiveness of the two techniques on Solution when explicit ranges are available.

Considering the efficiency of the two ATCG techniques, the overall results indicate that in general, the random techniques achieved testedness faster than the goal-oriented technique did initially, either with or without explicit range information. However, the difference between the techniques with explicit ranges is not as wide as without explicit ranges. With explicit ranges, the initial values that the goal-oriented technique begins to search with are closer to the values required for a useful the test case, so its search time is reduced. Moreover, the overall results also illustrate that, for the random technique, speed of generation slows after a short initial period; whereas for the goal-oriented technique, although speed of generation is initially slow, it continues to generates test cases steadily after the initial period.

Again examining Figure 4.4, we can see that the goal-oriented technique achieved more than 85% du coverage. This indicates that given enough time, the goal-oriented technique could exercise most of the feasible du pairs in a spreadsheet when explicit ranges were available. Thus, the du pairs remaining unexercised by the goal-oriented technique in that case are most likely be the infeasible du pairs of that spreadsheet if there are any. Based on this indication, if these results generalize, we could use the goal-oriented technique to approximately determine the infeasible du pairs, as long as we provide explicit ranges and enough time. It is more risky to use the random technique to do so, because in some spreadsheets it achieved lower than 70% du pair coverage.

These results support the following conclusions. We could automatically generate test cases that execute a large proportion of the feasible du pairs in a spreadsheet at the whole spreadsheet level, either with or without range information. Considering the effectiveness of generation, given no explicit ranges, the goal-oriented technique is the obvious choice; but given explicit ranges, the choice of which ATCG technique to apply is not so obvious. Considering the efficiency of generation, applying the random technique first for a short time and then applying the goal-oriented technique (with or without explicit ranges) could allow us to retain the effectiveness and improve efficiency at the same time.

Chapter 5

EMPIRICAL STUDIES TWO: DU PAIR LEVEL

In Chapter 4 we discussed experiments which evaluated two test case generation techniques at the whole spreadsheet level. As we discussed in Chapter 3, we are also interested in investigating our ATCG techniques at the du pair level. This chapter presents another group of experiments that address research questions at the du pair level. As in Chapter 4, we discuss some common issues first.

5.1 Common issues

5.1.1 Research questions

At the du pair level, we are interested in the following questions:

RQ3: Can we automatically generate test cases that execute a queried du pair with or without explicit range information?

RQ4: How do our test case generation techniques compare to each other in terms of effectiveness and efficiency at the du pair level with or without explicit range information?

5.1.2 Measurements

To measure the effectiveness of a ATCG technique at the du pair level, we used the proportion of the du pairs that the ATCG technique successfully generates test cases for over all the du pairs under test in the experiment. This indicates the likelihood that our ATCG techniques could generate a test case for a particular du pair. In these two experiments, we chose 300 seconds as a time limit. If an ATCG technique finds a test case within that time limit for a particular du pair, we consider it successful, otherwise, we consider it unsuccessful.

To measure the efficiency of a test case generator for a du pair, we use the time required in the generation procedure for that du pair.

5.1.3 Subjects and methods

To obtain du pairs on which to experiment, we used the same eight spreadsheets used in experiments 1A and 1B. We randomly picked 10 unique feasible du pairs from each spreadsheet to create our du pair subjects. Analysis of the 80 selected du pairs showed that 58.75% are p-use du pairs, while 41.25% are definition-c-use du pairs.

To reduce the influence of initial input values on the results, as in our previous experiments, our scripts executed our two test case generation techniques starting from the same random initial input values in each run, for a total of 35 runs for each selected du pair, with 35 different randomly generated initial values. Another point worth mentioning here is that, since our ATCG techniques continue until they have generated a test case for each subject du pair or timed out, it is not necessary to perform validations in these experiments.

In other respects, the procedures used in our scripts for these experiments are similar to those used in the experiments at the whole spreadsheet level.

5.2 Experiment 2A: with no explicit range information

5.2.1 Experiment design

This experiment is designed to evaluate the two techniques at the du pair level with no explicit range information. As in Experiment1A, we achieved this by using the smallest integer and the largest integer that the underlying system allowed as the default range.

The two independent variables manipulated in this experiment are:

- The eighty subject du pairs.
- The test case generation techniques: random and goal-oriented.

The dependent variable we measured is the total time required to generate a test case for a subject du pair. If the ATCG technique failed to generate a test case within the time limit, we consider the total run time for the ATCG technique as the measurement.

Similar to Experiment1A, this experiment was run using an 80×2 factorial design with 35 different initial input configurations per spreadsheet. For each subject du pair P, we applied two test case generation techniques starting from 35 randomly set initial inputs. This yielded 5600 data items for our analysis.

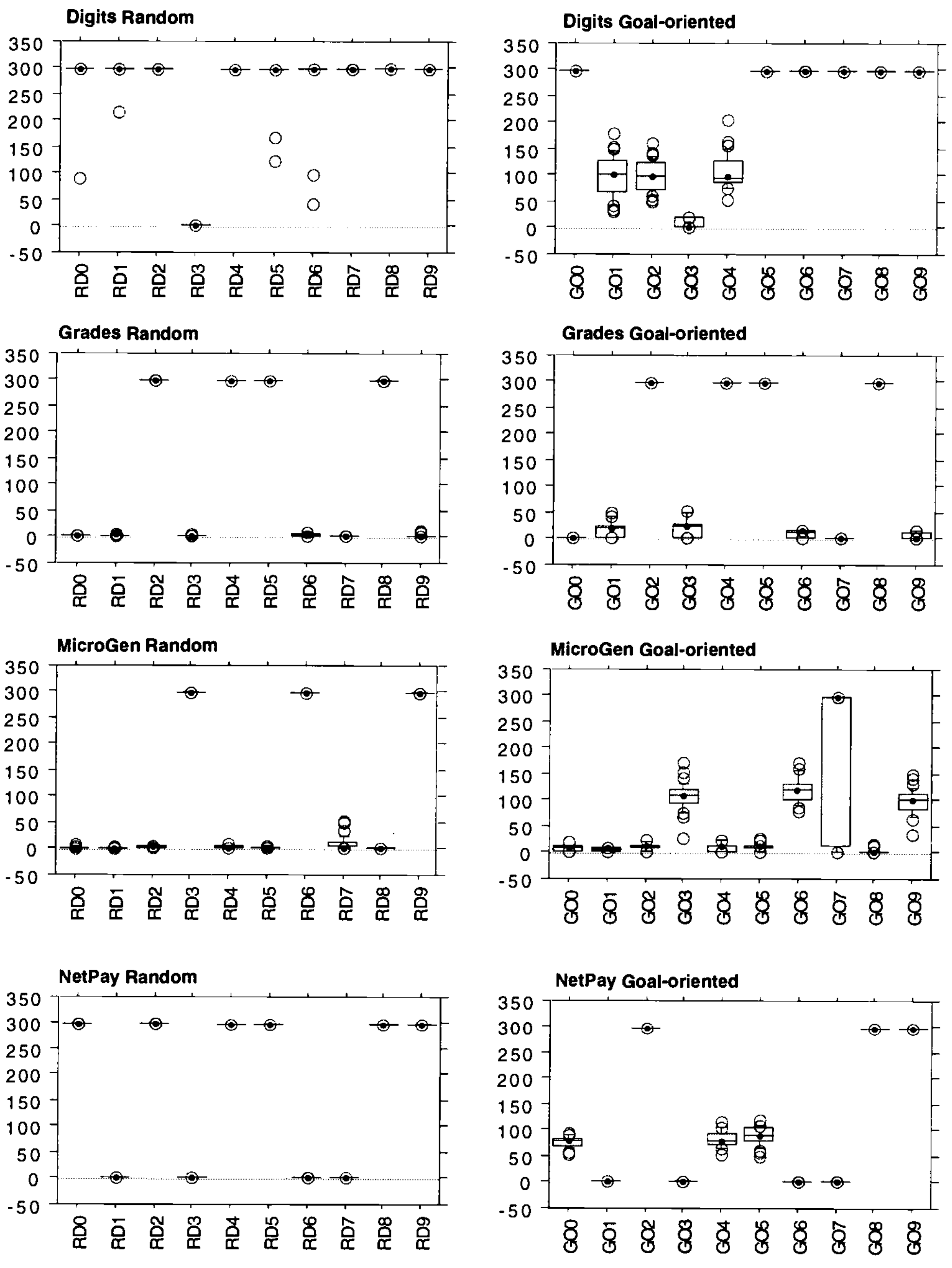


FIGURE 5.1: Box plots showing time (seconds) used in random and goal-oriented generation on subject du pairs in spreadsheets Digits, Grades, MicroGen and NetPay in Experiment 2A.

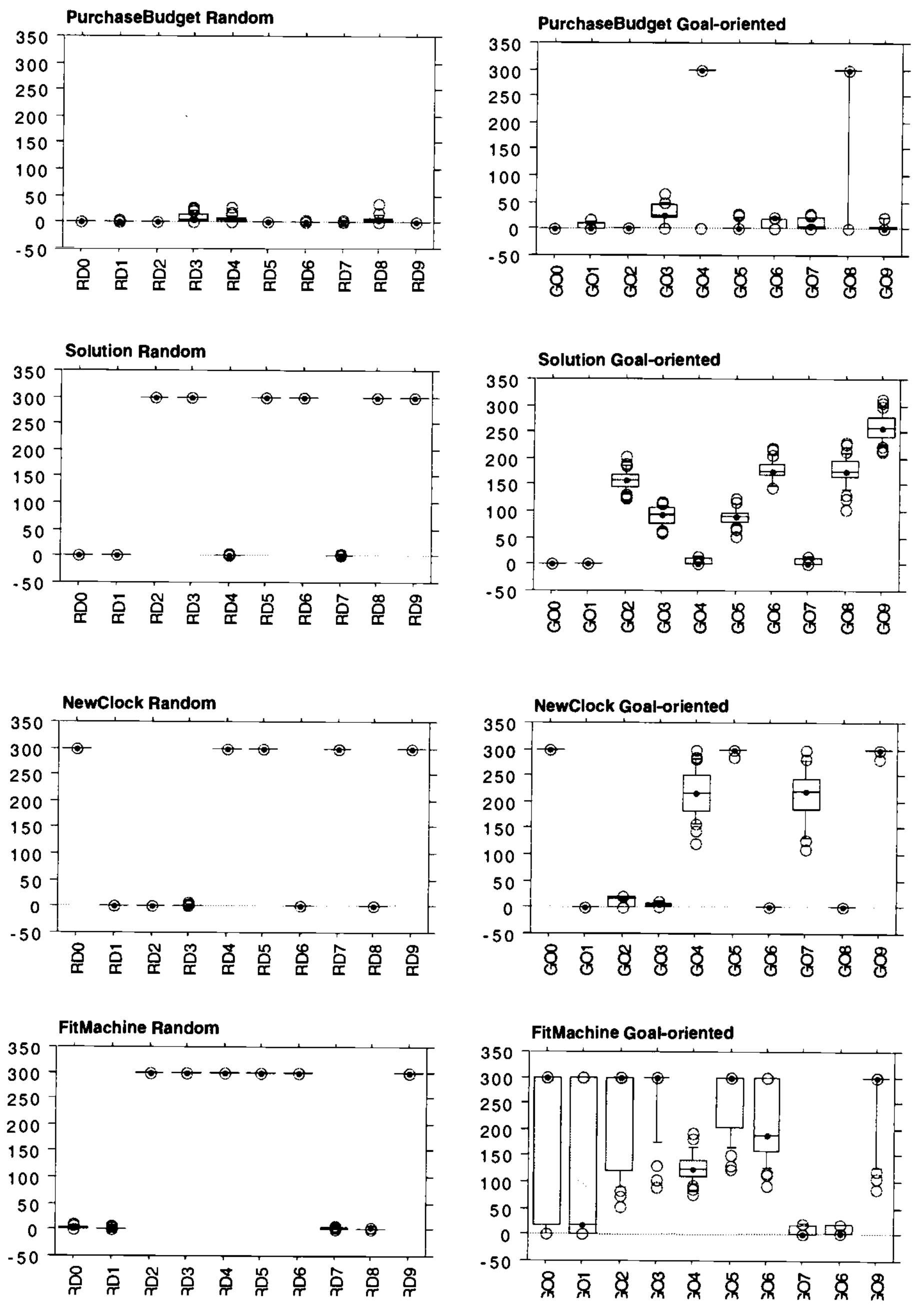


FIGURE 5.2: Box plots showing time (seconds) used in random and goal-oriented generation on subject du pairs in spreadsheets PurchaseBudget, Solution, NewClock and FitMachine in Experiment 2A.

5.2.2 Data and analysis

Figure 5.1 and 5.2 depict the effectiveness of the two techniques at the du pair level without explicit range information. Each figure shows four groups of box plots (each group is for a spreadsheet), including the box plots for the random technique and the goal-oriented technique in each group. The box plots illustrate the distribution of times spent on test generation by the two ATCG techniques for each subject du pair. The times vary from 0 seconds to 300 seconds. Data points at the 300 second mark indicate runs in which the ATCG technique failed to generate a test case for that du pair within the 300 second time limit. Data points appearing to be at the 0 second mark indicate runs in which test generation spent time close to 0 seconds.

For purpose of analysis, we can group results into 3 types, depending on where the data lies. Examining all the box plots, we see that (1) a technique can be always successful for a du pair over all 35 runs (no data point in a box plot for that du pair is located at the 300 second mark); (2) a technique can usually fail for a du pair over 35 runs (the median line in the box plot for that du pair is located at the 300 second mark); or (3) a technique can be partially successful for a du pair (there are some data points in the box plot located at the 300 second mark, but the median line in that box plot is below the 300 second mark). To differentiate these classes of du pairs we refer to them as *sp*, *fp* and *pp*, respectively. As the box plots show, for the random techniques, all du pairs are either *sp* or *fp*, whereas for the goal-oriented technique, all three types of du pairs exist.

The effectiveness data for our ATCG techniques is shown in Table 5.1. In this table, the number of *sp* du pairs, the number of *fp* du pairs, the number of *pp* du pairs and the total number of successful runs for both techniques, per

	random				goal-oriented			
spreadsheets	*sp*	*fp*	*pp*	successful runs	*sp*	*fp*	*pp*	successful runs
Digits	1	9	0	41	4	6	0	140
Grades	6	4	0	210	6	4	0	210
MicroGen	7	3	0	245	9	1	0	326
NetPay	4	6	0	140	7	3	0	245
PurchaseBudget	10	0	0	350	8	2	0	287
Solution	4	6	0	140	9	0	1	344
NewClock	5	5	0	175	5	3	2	245
FitMachine	4	6	0	140	3	5	2	198
Total%	51.3%	48.7%	0%	51.5%	63.8%	30%	6.2%	71.3%

TABLE 5.1: Data about effectiveness of the two ATCG techniques at the du pair level with no explicit range information.

spreadsheet, are listed. The last row in the table shows the proportion of three types of du pairs over all subject du pairs and the percentage of successful runs over the $8 \times 10 \times 35 = 2800$ runs for each technique. As the table indicates, the random technique resulted in fewer *sp* du pairs than the goal-oriented technique for most spreadsheets: over the eight spreadsheets, the proportion of total *sp* du pairs over all subject du pairs of the random technique was 51.3%, while it was 63.8% for the goal-oriented technique. Also, there are fewer *fp* du pairs for the goal-oriented technique than for the random technique: the proportion of total *fp* du pairs over all subject du pairs for the random technique was 48.7%, while it is only 30% for the goal-oriented technique. In addition, although the goal-oriented technique obtained no more *sp* du pairs than the random technique on FitMachine and NewClock, there are some *pp* du pairs of the goal-oriented technique on those spreadsheets. Considering all 39 du pairs that were *fp* for the random technique, 14 of these were *sp* du pairs for the goal-oriented technique and 4 of them were *pp* du pairs for the goal-oriented technique; considering all 24 du pairs that were *fp* for the goal-oriented technique, only 3 of them were successfully covered by the random technique. Moreover, the percentages of successful runs for the two techniques indicates that the goal-oriented technique achieved more successful runs overall than the random technique in this experiment.

Again observing Figure 5.1 and Figure 5.2, it is interesting that although the random technique failed to generate test cases for more du pairs than the goal-oriented technique, the time needed for the random technique to cover a du pair when it was successful was less than 10 seconds in most cases. On the other hand, although the goal-oriented technique covered some du pairs that the random technique could not cover, the time it required was often more than 100 seconds. In addition, comparing the du pairs that are successfully covered by both two techniques, the random technique always required less time than the goal-oriented technique. This indicates that the goal-oriented technique is not as efficient as the random technique at the du pair level in general. This also explains why the goal-oriented technique is always slower initially than the random technique at the whole spreadsheet level.

5.3 Experiment 2B: with explicit range information

5.3.1 Experiment design

We performed a second experiment to assess the effect of using explicit range information at the du pair level. We used the same subjects as in Experiment 2A and the same range information used in experiment1B.

	random				goal-oriented			
spreadsheets	*sp*	*fp*	*pp*	successful runs	*sp*	*fp*	*pp*	successful runs
Digits	9	1	0	315	4	0	6	281
Grades	8	1	1	312	7	2	1	271
MicroGen	7	2	1	277	9	1	0	325
NetPay	10	0	0	350	3	0	7	288
PurchaseBudget	10	0	0	350	8	1	1	315
Solution	4	6	0	150	10	0	0	350
NewClock	10	0	0	350	7	1	2	325
FitMachine	8	0	2	344	3	0	7	278
Total%	82.5%	12.5%	5%	87.4%	63.8%	6.2%	30%	86.9%

TABLE 5.2: Data about effectiveness of the two ATCG techniques at the du pair level with explicit range information

5.3.2 Data and analysis

Similar to the corresponding figures in Experiment 2A, Figures 5.3 and 5.4 depict the effect of the two ATCG techniques at the du pair level with explicit range information. Through observing the box plots we classified the three types of du pairs as we did in Experiment 2A. Table 5.2 shows the data about the three types of du pairs for both ATCG techniques.

Examining Table 5.2, we find that both techniques improved their effectiveness with explicit ranges. However, the degrees of improvement are not the same. The proportion of *sp* du pairs over all subject du pairs increased to 82.5% for the random technique, while it remained at 63.8% for the goal-oriented technique. Also, the proportions of *fp* du pairs for both ATCG techniques decreased to 12.5% and 6.2%, respectively. In addition, with explicit ranges, the number of *pp* du pairs for the random technique increased a little (from zero to 5%) but this number increased a lot for the goal-oriented technique (from 6.2% to 30%).

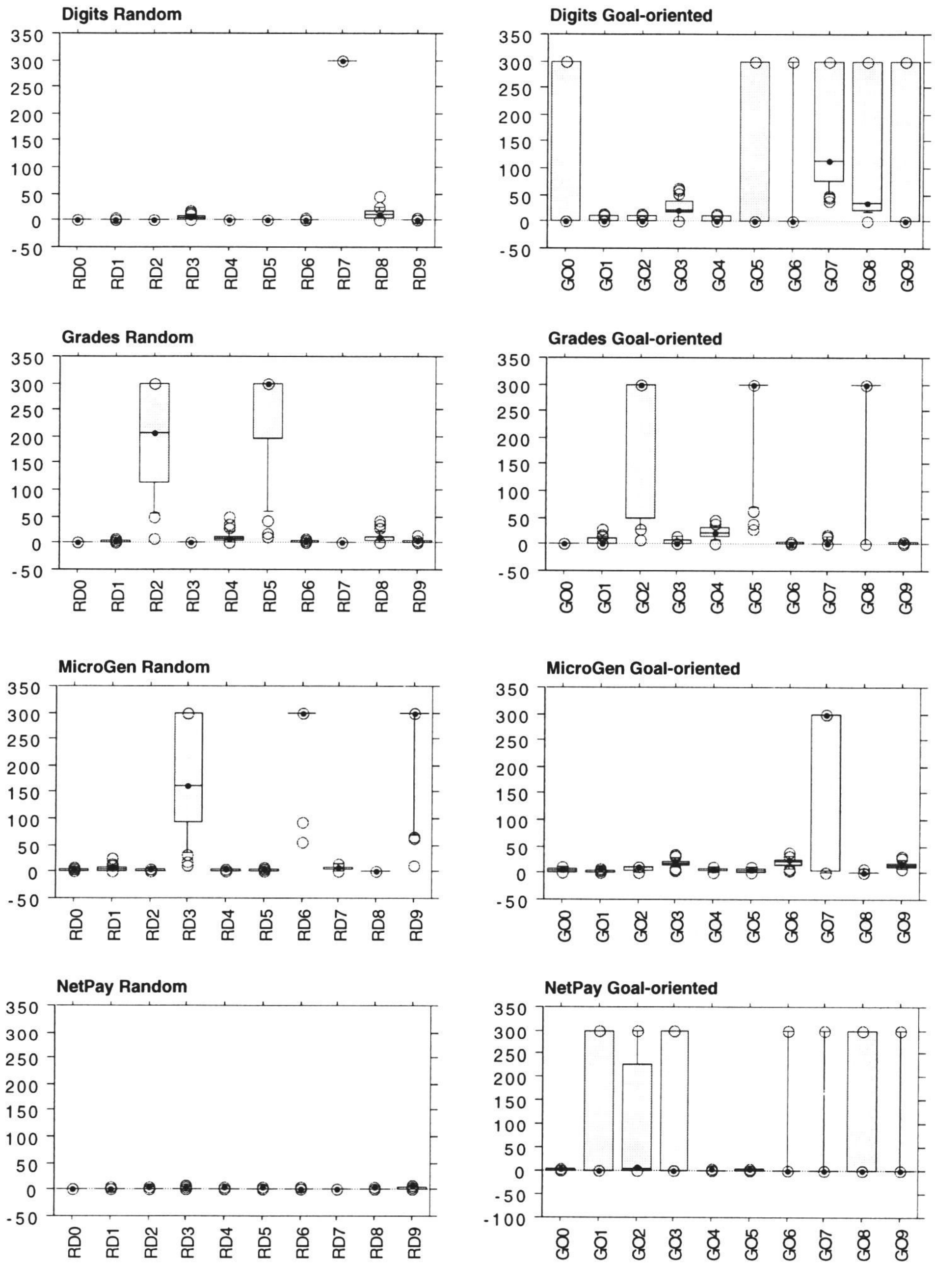


FIGURE 5.3: Box plots showing time (seconds) used in random and goal-oriented generation on subject du pairs in spreadsheets Digits, Grades, MicroGen and NetPay in Experiment 2B.

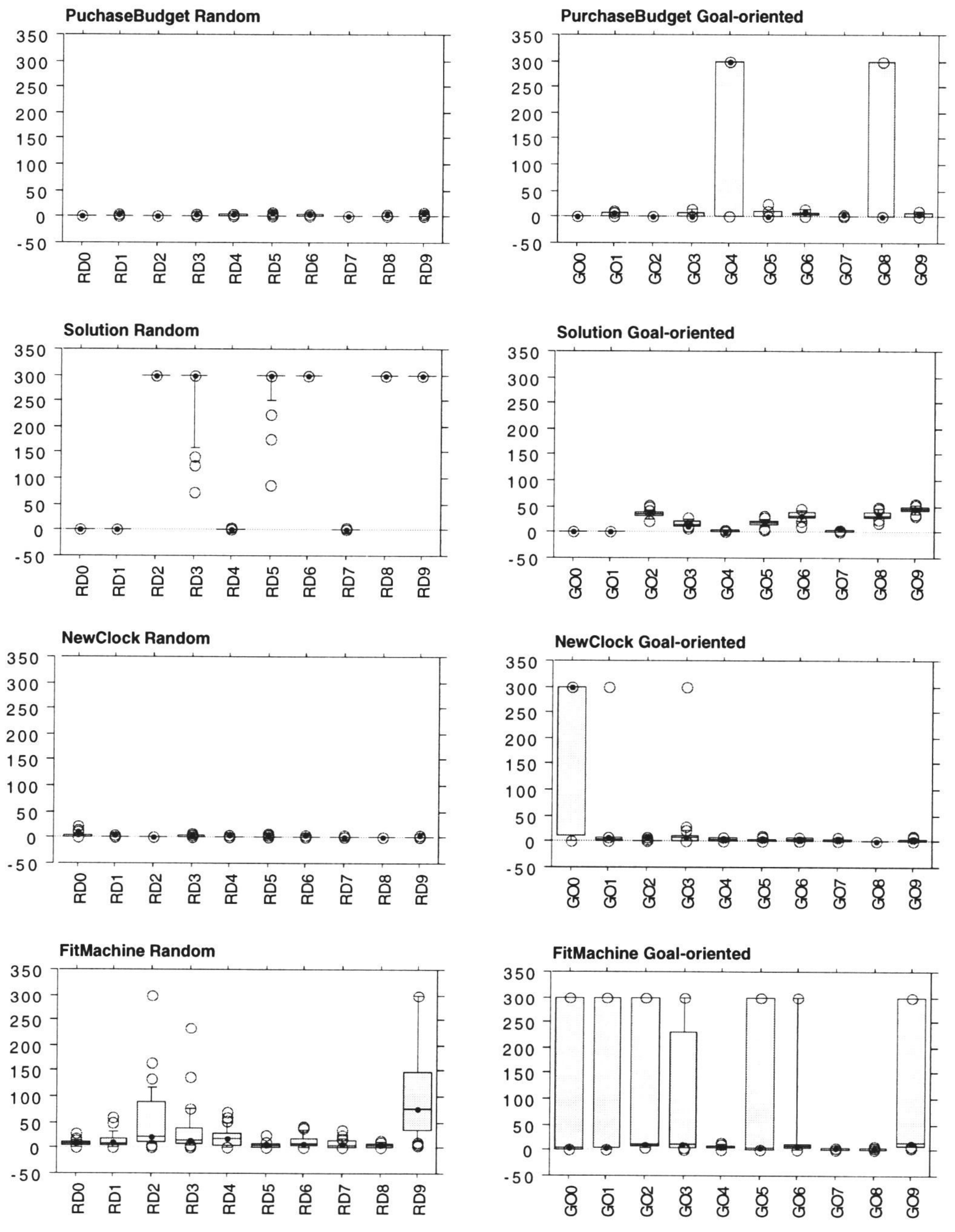


FIGURE 5.4: Box plots showing time (seconds) used in random and goal-oriented generation on subject du pairs in spreadsheets PurchaseBudget, Solution, NewClock and FitMachine in Experiment 2B.

Although the number of *sp* du pairs for the goal-oriented technique is less than that for the random technique, only 6.2% of the du pairs were *fp* du pairs for the goal-oriented technique. Observing the box plots for the 24 *pp* du pairs for the goal-oriented technique, we find that the median for most of those du pairs was less than 10. This indicates that these 24 du pairs were more often than not successfully exercised by the goal-oriented technique. This also indicates that although the goal-oriented technique did not make progress on the proportion of its *sp* du pairs, it did make progress on effectiveness overall. Adding the total number of *sp* du pairs and *pp* du pairs for the goal-oriented technique produces a higher percentage than for the random technique. Moreover, the total number of successful runs for both techniques are close. Thus, overall, it is difficult to tell which of the two ATCG techniques is more likely to exercise an arbitrary du pair when explicit ranges are available.

Considering speed of generation, the goal-oriented technique made noticeable progress with explicit range information. The overall goal-oriented generation time was less than 50 seconds when the goal-oriented technique was successful on a particular du pair. However, the speed of random generation was still higher than that of goal-oriented generation in most cases.

5.4 Discussion

It is worth noting that the goal-oriented technique failed to cover 21% of the du pairs at the du pair level while it exercised most du pairs at the whole spreadsheet level, when explicit ranges are provided. One possible explanation for this is that the goal-oriented technique deals with only an isolated du pair at the du pair level, while at the spreadsheet level it deals with a group of related du pairs. As we discussed in Section 3.3.4, directly generating a test case for a du pair deep in a chain of dependencies can be relatively difficult. At the whole spreadsheet level, since any one of the unvalidated du pairs in a spreadsheet is considered as a subgoal in the goal-oriented generation, the "easier" du pairs are likely to be exercised first. Exercising those du pairs will also cause some of the "deep" du pairs to be exercised at the same time, or make some other deep du pairs easier to exercise.

Overall, it is possible to conclude that, when generating test cases for specific du pairs, and when no explicit ranges are provided, the goal-oriented technique has a greater chance of exercising an arbitrary du pair than the random technique, as long as enough time is provided. When explicit range information is available, the choice is not so obvious.

Chapter 6

EMPIRICAL STUDIES THREE: LARGE SPREADSHEETS

As we discussed in Chapter 4, the subject spreadsheets used in our initial experiments are of small and medium size. Larger, more complex spreadsheets may be subject to different cost-effectiveness trade-offs. To address this threat, in this study, we repeat the same experiments reported in the preceding chapters, except that in this study the subject spreadsheets are large spreadsheets.

6.1 Subjects

For this study, two large spreadsheets were created by an experienced Forms/3 user. Some data about the two large spreadsheets is shown in Table 6.1. The smaller of the two spreadsheets, RandomJury, determines statistically whether a panel of jury members was selected randomly. Another spreadsheet, MBTI, implements a version of the Myers-Briggs Type Indicator (a personality test). Given integer answers to twenty questions, this spreadsheet tallies the scores and reports personality types. Examining the formulas of the two large spreadsheets, although RandomJury has fewer du pairs, its cell reference level is deeper than that of MBTI. However, the proportion of predicates over all expressions in MBTI is higher than that in RandomJury.

spreadsheets	No. of cells	No. of du pairs	No. of feasible du pairs	No. of expressions	No. of predicates
RandomJury	29	266	188	93	32
MBTI	48	784	780	248	100

TABLE 6.1: Data about large experimental subjects

6.2 Whole spreadsheet level experiments

As in study one, we performed two experiments to investigate the effectiveness and efficiency of our two ATCG techniques on larger spreadsheets, at the whole spreadsheet level. The first experiment is performed without explicit range information and the second one is performed with explicit range information. Since both spreadsheets are large spreadsheets, we performed several trial runs for each large spreadsheet to obtain the proper time limits. Results suggested that 5000 seconds was sufficient for RandomJury and 10000 seconds was sufficient for MBTI. Remember that these times are limits on the experiments, not the times we would expect a user to wait.

6.2.1 Experiment 3A: with no explicit range information

The design of this experiment is the same as that of Experiment 1A. Figure 6.1 illustrates the average cumulative testedness of the two large spreadsheets achieved over time when no explicit ranges are provided.

The random technique exercised a large percentage of du pairs in RandomJury and the goal-oriented technique exercised most du pairs in MBTI.

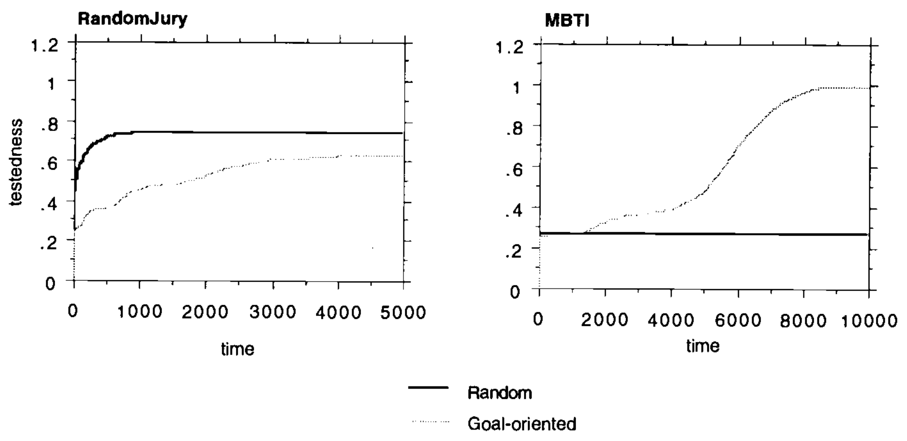


FIGURE 6.1: Test case generation efficiency on large spreadsheets at the whole spreadsheet level with no explicit range information provided. Graphs show the average cumulative testedness (ranging from 0.0 to 1.0) over 1200 seconds on eight subjects.

However, the random technique only achieved 0.2564 testedness on MBTI and the goal-oriented technique achieved only 0.63 testedness on RandomJury. Examination of the formulas of MBTI shows that MBTI contains many complex predicates with "equal" operators. This supports our conjecture that the formulas in spreadsheets could affect the effectiveness of the random technique. In addition, the deep cell references in RandomJury also may explain why the goal-oriented technique achieved lower testedness on RandomJury, supporting our conjecture about the depth of du pairs affecting the effectiveness of the goal-oriented technique. However, the overall results indicate that the goal-oriented technique is more capable than the random technique of exercising a large part of the du pairs in a spreadsheet when no explicit ranges are provided.

Considering the speed of test generation, on RandomJury, the goal-oriented technique achieved testedness slower than the random technique did initially. On MBTI, the random technique did not make any progress after the first 10

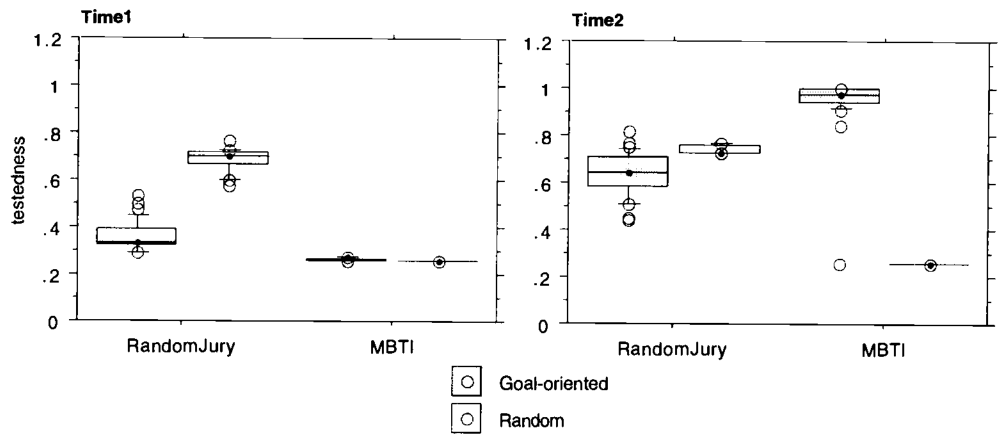


FIGURE 6.2: Test case generation effectiveness on large spreadsheets at the whole spreadsheet level with no explicit range information provided. Boxplots show the distribution of testedness (ranging from 0.0 to 1.0) at two times. For each spreadsheet, two box plots are given: the left plot depicting data for the goal-oriented technique, the right plot depicting data for the random technique.

seconds. While the goal-oriented technique was slow initially, it continued to generate test cases, speeding up in the later period of generation and ending at almost 100% du pair coverage. This observation again confirms that the random technique may be faster than the goal-oriented initially, but later it slows down while the goal-oriented technique continues to make progress.

As in our first experiment, we selected two times for each spreadsheet to investigate the distribution of the testedness over the 35 runs and compare the success of generation in the initial period and later during the generation period. The box plots are shown in Figure 6.2. We also performed paired t-tests to confirm the differences in testedness observed for the two techniques at two different times. The data shown in this table confirms our observations about Figures 6.1 and 6.2. Overall, the results indicate that the effectiveness and the efficiency of both ATCG techniques without explicit range information are

	Time 1			Time 2		
spreadsheets	Mean Diff.	t-Value	P-Value	Mean Diff.	t-Value	P-Value
RandomJury	-.322	-26.687	<.0001	.097	-5.959	<.0001
MBTI	-.009	8.795	<.0001	-.708	117.513	<.0001

TABLE 6.2: Paired t-test for Experiment 3A

consistent with our conclusion made in the corresponding experiment in study one on small spreadsheets.

6.2.2 Experiment 3B: with explicit range information

We designed this experiment to be similar to Experiment 1B. We obtained explicit range information for our large spreadsheets in the same manner described that study. Figure 6.3 shows the results of this experiment. Similar to Figure 6.1, Figure 6.3 illustrates the average cumulative testedness when providing explicit range information for the input cells.

Examining Figures 6.1 and 6.3, we see that the use of explicit ranges noticeably affected the effectiveness of the random technique in terms of final testedness on MBTI (increasing from .2564 to 0.9958), and noticeably affected the initial effectiveness of the goal-oriented technique on MBTI. However, these influences are not obvious on RandomJury. Overall, both techniques exercised more than two thirds of the du pairs in the two large spreadsheets, and the goal-oriented technique was not as efficient as the random technique initially. Figure 6.4 and Table 6.3 support our observations in terms of efficiency. Overall the results affirm our previous conclusion about effectiveness and efficiency

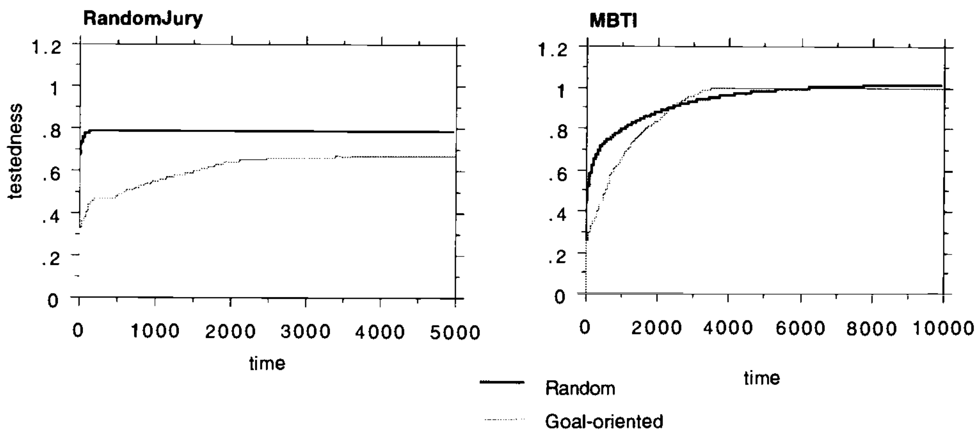


FIGURE 6.3: Test case generation efficiency on large spreadsheets at the whole spreadsheet level with explicit range information provided. Graphs show the average cumulative testedness (ranging from 0.0 to 1.0) over 1200 seconds on eight subjects.

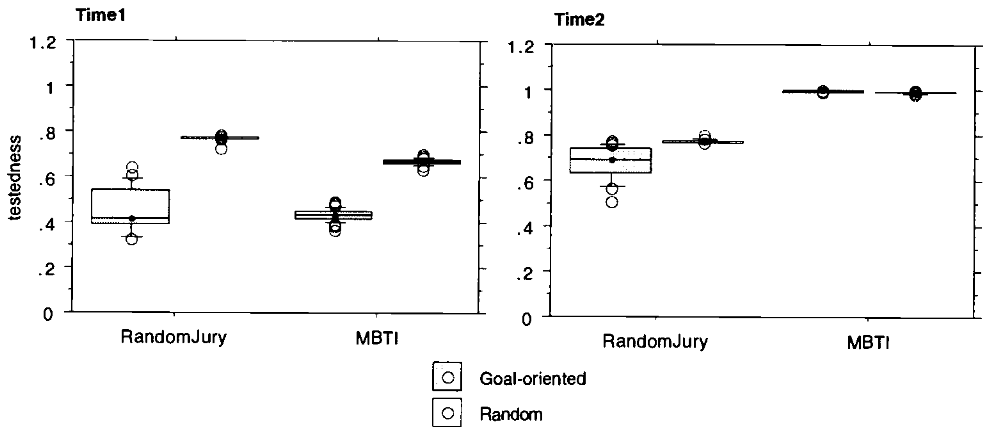


FIGURE 6.4: Test case generation effectiveness on large spreadsheets at the whole spreadsheet level with explicit range information provided. Boxplots show the distribution of testedness (ranging from 0.0 to 1.0) at two times. For each program, two box plots are given: the left plot depicts data for the goal-oriented technique, the right plot depicts data for the random technique.

when explicit ranges are provided for both techniques on small spreadsheets.

	Time 1			Time 2		
spreadsheets	Mean Diff.	t-Value	P-Value	Mean Diff.	t-Value	P-Value
RandomJury	-.306	-19.415	<.0001	.098	-8.275	<.0001
MBTI	-.233	-51.396	<.0001	-.005	6.063	<.0001

TABLE 6.3: Paired t-test for Experiment 3B

6.3 Du pair level experiments

The two experiments described next are designed to examine the effectiveness and efficiency of our two ATCG techniques for du pairs selected from large spreadsheets. We randomly selected 10 du pairs from each large spreadsheet. The first experiment is designed in the same way as Experiment 2A, with no explicit ranges, and the second experiment is designed in the same way as Experiment 2B, with explicit ranges.

Figures 6.5 and 6.6 illustrate the results of the experiments without and with explicit range information respectively. The box plots in the two figures show that there was no obvious difference between the effectiveness of both techniques at the du pair level on large spreadsheets. The results with explicit ranges are consistent with those in the experiment with explicit ranges in Chapter 5. However, without explicit ranges, the goal-oriented technique did not obtain better effectiveness than the random technique as in the corresponding experiment in Chapter 5. We suggest the following possible reasons: 1) there are about 1000 du pairs in these two large spreadsheets, the 20 subject du pairs used in this study may not be representative; 2) Since the large spreadsheets have deep cell

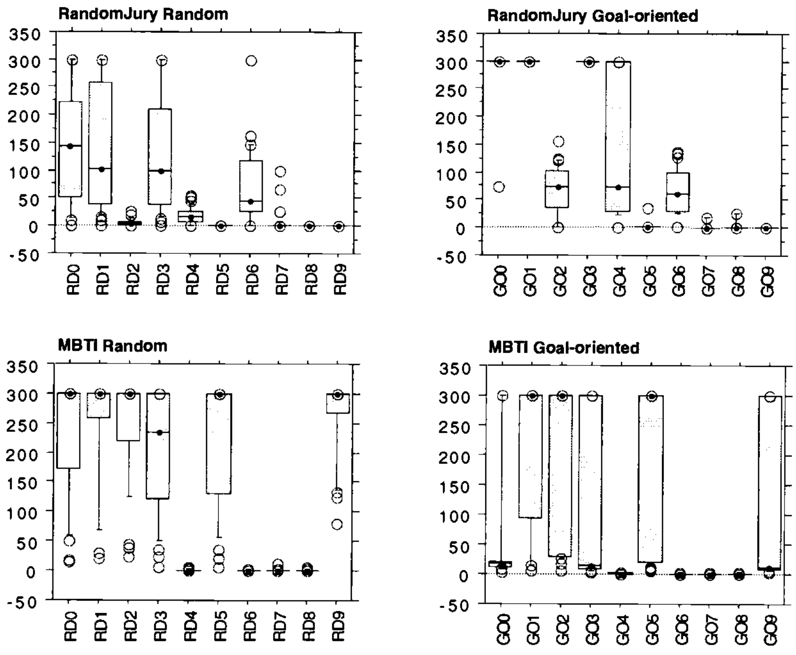


FIGURE 6.5: Box plots show time (seconds) used in random and goal-oriented test case generation for subject du pairs selected from large spreadsheets, with no explicit range information.

references and more complex formulas, the depths of du pairs selected from them are likely be greater, on average, than in the previous study. This might affect the goal-oriented technique more than the random technique. Additional studies could address these threats by using more du pairs selected from more large spreadsheets as subjects, and improving the goal-oriented technique to deal with du pairs involving deep level cell references.

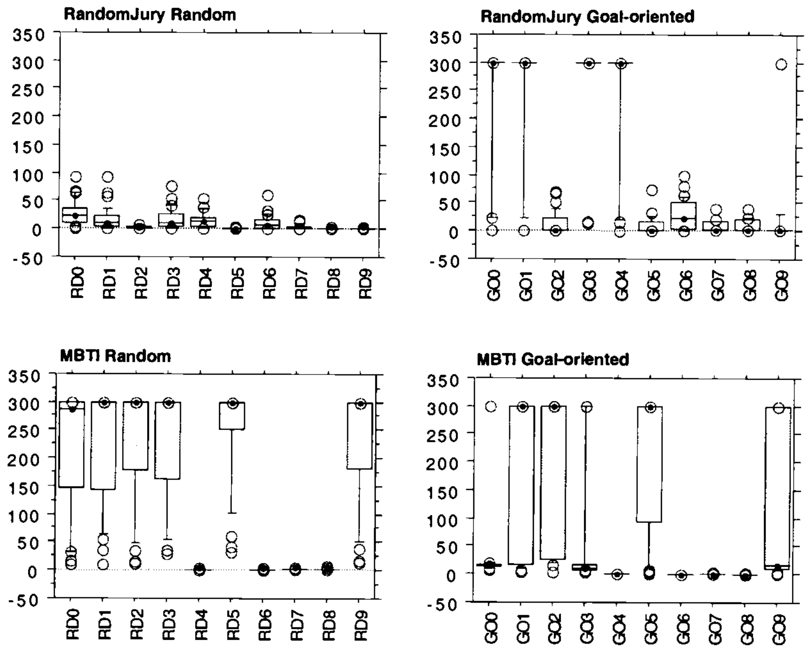


FIGURE 6.6: Box plots show time (seconds) used in random and goal-oriented test case generation for subject du pairs selected from large spreadsheets, with explicit range information.

Chapter 7

CONCLUSION AND FUTURE WORK

In this thesis, we have presented an automatic test case generation methodology for spreadsheet languages. Our methodology uses an incremental generation strategy, and is driven by the end user's request. It could help end users generate test case at three levels: the whole spreadsheet level, the cell level and the du pair level. In addition, our methodology is integrated with the highly interactive spreadsheet programming environment, presenting test data visually. The underlying ATCG technique for spreadsheets has been developed by properly adapting an appropriate existing technique for imperative programs. The details of the underlying ATCG techniques do not need to be known by the end users. We prototyped our ATCG techniques in the research spreadsheet language Forms/3.

To assess our methodology, we performed several empirical studies to investigate the effectiveness and efficiency of the two ATCG techniques at the whole spreadsheet level, at the du pair level, and for large spreadsheets, respectively. Our result indicate that, at both the whole spreadsheet level and the du pair level, the goal-oriented technique is more effective than the random technique in general when no explicit ranges are provided, whereas the differences are not obvious when explicit ranges are used. The results also indicate that the random technique is more efficient than the goal-oriented technique.

Overall, using range information could improve the effectiveness of the random technique and improve the efficiency of the goal-oriented technique.

The initial results obtained in these empirical studies suggest the following future work:

- As our initial results indicate, when no explicit ranges are provided, the goal-oriented technique has greater effectiveness and the random technique has a greater efficiency in general. Considering these results, a proper combination of these two techniques might obtain both effectiveness and efficiency.

- As we discussed earlier, there are several threats to external validity that affect our ability to generalize results. These threats could be addressed only through additional studies that use other spreadsheets, and use user-provided range information and initial values.

- We are planning to improve the goal-oriented technique in order to enhance its searching ability on du pairs that are deep in dependence chains. As described in [11], a chaining approach, which utilizes the data dependence analysis to assist the search process, can improve the effectiveness of the goal-oriented technique. We will integrate this chaining approach into our methodology.

- As our initial results in Experiment 1B suggest, using the goal-oriented technique can approximately determine the infeasible du pairs in a spreadsheet when explicit ranges are provided. Additional studies of this effect on more subject spreadsheets are necessary.

- Our empirical studies have focused on test case generation at the whole spreadsheet level and the du pair level. We plan to conduct additional studies at the cell level.

- Since our final goal is to help end users generate test cases in spreadsheet testing, end-user studies are necessary, in order to assess whether users can effectively employ our ATCG methodology.

We hope that through the work reported in this thesis, and the future work listed above, we can provide an automatic test case methodology for spreadsheet languages, which will help end users test their spreadsheets more effectively and more efficiently.

BIBLIOGRAPHY

[1] A. Ambler, M. Burnett, and B. Zimmerman. Operational versus definitional: A perspective on programming paradigms. *Computer*, 25(9):28–43, September 1992.

[2] A. Bertolino and M. Marre. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–898, December 1994.

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

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

[5] M. Burnett and H. Gottfried. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction*, pages 1–33, March 1998.

[6] M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang. Toward visual programming languages for steering in scientific visualization: a taxonomy. *IEEE Computing Science and Engineering*, 1(4), 1994.

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

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

[9] R. DeMillo and A.J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.

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

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

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

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

[14] B.F. Jones, H.H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, pages 299–306, September 1996.

[15] B. Korel. A dynamic approach of automated test data generation. In *the Proceeding of the Conference on Software Maintenance.*

[16] B. Korel. Automated test data generation for programs with procedures. In *the Proceeding of the International Symposium on Software Testing and Analysis.*

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

[18] B. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *ACM CHI '91*, pages 243–249, April 1991.

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

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

[21] R. Panko and R. Halverson. Spreadsheets on trial: A survey of research on spreadsheet risks. In *the Proceedings of the 29th Hawaii International Conference on System Sciences*, January 1996.

[22] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[23] J. Reichwein, G. Rothermel, and M. Burnett. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. In *The 2nd Conference on Domain Specific Languages (DSL '99)*, pages 25–38, October 1999.

[24] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and Andrei Sheretov. A methodology for testing spreadsheets. Technical Report TR: 99-60-02, Oregon State University, January 1999.

[25] G. Rothermel, L. Li, and M. Burnett. Testing strategies for form-based visual programs. In *The 8th International Symposium on Software Rellia-bility Engineering*, pages 96–107, November 1997.

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

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

[28] G. Viehstaedt and A. Ambler. Visual representation and manipulation of matrices. *Journal of Visual Languages and Computing*, 3(3):273–298, September 1992.

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

[30] Edin Zulic. Test input generation. Technical report, Oregon State University, December 1998.