DISSERTATION

Spreadsheet Testing Using Interval Analysis

von

Yirsaw Ayalew

Ausgeführt zum der Erlangung des akademischen Grades Doktor der technischen Wissenschaften

Institut für Informatik-Systeme
der Universität Klagenfurt
Fakultät für Wirtschaftswissenschaften und Informatik

begutachtet von:
o.Univ. Prof. Dr. Roland Mittermeir
o.Univ. Prof. Dr. Johann Eder

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, daß ich die vorliegende Schrift verfaßt und alle ihr vorausgehenden oder sie begleitenden Arbeiten durchgeführt habe. Die in der Schrift verwendete Literatur sowie das ausmaß der mir im gesamten Arbeitsvorgang gewährten Unterstützung sind ausnahmslos angegeben. Die Schrift ist noch keiner anderen Prüfungsbehörde vorgelegt worden.

Yirsaw Ayalew

Klagenfurt, November, 2001

Dedication

Dedicated to my father, Ayalew Alemneh, who always encourages me to learn.

Acknowledgements

I would like to specially thank my advisor, Prof. Roland Mittermeir, for his very helpful guidance, advice, discussion, and encouragement over the period of my study. Not only in the dissertation, I have also learnt a lot from him during the various courses I took with him. I would like also to thank Prof. Johann Eder, my second advisor, for his important comments while reading the thesis. I am greatly indebted to the research group of Prof. Roalnd Mittermeir for their constructive comments and ideas during our weekly seminar discussions in the privatissimum.

My thanks goes to the Austrian Academic Exchange Service for providing financial support for the whole period of my study.

My thanks are due to senior undergraduate students Agim Rama, Margarita Hainschitz, and Tassilo Simon for their participation in the development of the prototype tool during their practical course.

Finally, I would like to thank all my friends for their continual encouragement during the challenging period of my study.

Abstract

Spreadsheet programs, artifacts developed by non-programmers, are used for a variety of important tasks and decisions. Yet a significant proportion of them have severe quality problems. This thesis presents a new approach for checking spreadsheets on the premises that their developers are not software professionals. The approach takes inherent characteristics of spreadsheets as well as the conceptual models of spreadsheet programmers into account and incorporates ideas from symbolic testing and interval analysis.

Unlike symbolic testing, which requires expressing formulas in terms of input variables, interval-based testing uses intermediate variables for the purpose of narrowing down computed intervals. In addition, while symbolic testing is used to validate a formula for any possible values of the input variables, interval-based testing requires the values of the variables to be expressed as intervals and validity is determined based on the intervals provided.

The observation that spreadsheets are mainly used for numerical computations enables us to introduce the idea of interval analysis to spreadsheet testing. Interval-based testing focuses on the functionality of spreadsheet formulas instead of the internal structure of a spreadsheet program (i.e., it is not based on code coverage criterion). It requires the user to specify input and expected intervals for desired input and formula cells respectively. This will be documented in a behind-the-scene spreadsheet and used to perform interval computations during the verification of a given spreadsheet. In addition, the expected intervals provided by the user are verified for reasonableness using interval analysis. The approach provided is thus essentially a kind of stratified plausibility check based on the consistency of legitimate boundaries users might specify for computations.

Kurzfassung

Tabellenkalkulationen (Spreadsheets) werden von Anwendern ohne Programmierkenntnisse entwickelt. Sie werden für eine Vielzahl von wichtigen Aufgaben benötigt und dienen als Grundlage für Entscheidungen. Es stellt sich jedoch heraus, dass ein Großteil der eingesetzten Tabellenkalkulationen schwerwiegende Qualitätsmängel hat. In dieser Arbeit wird ein neuartiges Prüfverfahren für Tabellenkalkulationen präsentiert, die von Anwendern ohne Programmierkenntnissen entwickelt wurden. Dabei werden sowohl inhärente Tabellenkalkulationen-Charakteristiken, als auch konzeptuelle Modelle von Tabellenkalkulationsentwicklern berücksichtigt und Ideen aus den Bereichen des symbolischen Testens und der Intervallanalyse miteinbezogen.

Im Gegensatz zum symbolischen Testen, das Formeln auf der Basis von Eingabevariablen erfordert, werden bei Intervall-basiertem Testen Zwischenergebnisse zur Einschränkung der berechneten Intervalle verwendet. Weiters wird bei symbolischem Testen eine Formel für jeden möglichen Wert der Eingabevariablen validiert. Dies ist bei Intervall-basiertem Testen nicht der Fall. Hier werden die Variablen als Intervalle ausgedrückt und die Gültigkeit der Variablen auf der Basis dieser Intervalle bestimmt.

Hauptsächlich werden Tabellenkalkulationen für numerische Berechnungen eingesetzt. Diese Tatsache rechtfertigt den Einsatz der Technik der Intervallanalyse für das Testen von Tabellenkalkulationen. Intervallanalyse zielt auf das Testen der Funktionalität von Spreadsheet-Formeln und nicht auf die interne Struktur der Tabelle ab (so z.B. wird Code-Überdeckung nicht geprüft). Der Anwender muß die Eingabewerte und auch die Intervalle für die Eingabe bzw. Zellformeln spezifizieren. Diese Informationen werden in einem weiteren (versteckten) Spreadsheet gespeichert und dienen während der Verifikation zur Berechnung der gültigen Intervalle. Zusätzlich werden die Erwartungswerte der vom Benutzer definierten Intervalle mittels Inter-

vallanalyse auf Plausibilität geprüft. Damit ist dieses Verfahren hauptsächlich ein mehrschichtiger Ansatz zur Plausibilitätprüfung von Tabellenkalkulationen, das auf der Konsistenz von gültigen, von Anwendern spezifizierten Grenzwerten basiert.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter: 1
Introduction

1.1 Introduction

To ensure the quality of software, there are a variety of techniques and tools used at various stages of the development process. Among these techniques, testing is a very crucial activity to ensure the quality of software products and as a result up to 50% of the cost of development is devoted to testing [7, 71, 109].

To aid testers, a variety of techniques have been proposed and are being used at various stages of development. For end-user programs which are often developed without any formal development methodology, the error rate is high, thus demanding for a suitable testing methodology. Boehm and Basili [9] have indicated the necessity of providing defect detection tools for end-user programmers whose number is growing rapidly. One of the most commonly used end-user programming environments is the spreadsheet system.

Spreadsheet systems are widely used and highly popular end-user systems. They are used for a variety of important tasks such as mathematical modelling, scientific computation, tabular and graphical data presentation, data analysis and decision making. They have actually contributed a lot to the promotion of end-user computing. Millions of people use spreadsheets everyday and they are the choices of many individuals and companies. Many business applications are based on the results of spreadsheet computations and as a result important decisions are made based on spreadsheet results. The provision of computational techniques that match users' tasks makes programming easier. Besides their use for numerical computations which is what they were initially designed for, spreadsheet models are also found to be very important in other areas too. The computational models of spreadsheets are adopted in areas such as information visualization, concurrent computation, user interface specifications to name a few. Such extensions indicate the advantages of spreadsheets as a model and mechanism for various kinds of programming environments. There is also a trend in using the spreadsheet model as a general model for end-user programming [73].

Despite their popularity due to their ease of use and suitability for numerical computations, a significant proportion of spreadsheet programs have severe quality problems. In recent years, there has been an increased awareness of the potential impact of faulty spreadsheets on business and decision making. A number of experimental studies and field audits [10, 86, 85, 81, 82, 87, 84, 83] have already showed the serious impact spreadsheet errors have on business and on decisions made based on spreadsheet programs. Moreover, the developers of spreadsheet programs are mainly end-users who are not expected to follow the formal process of software development and as a result their reliance on the initial correctness of their programs is overly high. In addition, users do not have effective methods of detecting the existence of faults.

Though spreadsheets are software too which are developed by end users, they are also somewhat different. Therefore, traditional software quality assurance techniques are not directly applicable or are applicable only to a limited extent to improve spreadsheet quality. This is also true taking into consideration the expertise of users. Some approaches have been proposed to tackle the spreadsheet quality problem by using preventive mechanisms such as design methodology. These approaches are preventive which aid to avoid some faults before they are introduced into programs, but they do not provide mechanisms of detecting latent faults. Other approaches propose the adaptation of software engineering approaches to spreadsheets. These approaches are quite appealing considering spreadsheets as computer programs. However, the developers of spreadsheets are not trained in software engineering theory; they are end-users who are not bothered about the formal process of software development. Therefore, these approaches have a limited applicability.

Furthermore, users will not accept methodologies which distract their freedom of work with which they are comfortable. In suggesting the use of software engineering principles to spreadsheet development, the adaptation should take a serious consideration of the spreadsheet developers. Unless the conceptual model of the spreadsheet developer is incorporated in the process, all those nice looking approaches may not be practiced by the large community of end users. On the other hand, spreadsheet programs are different from conventional programs. Therefore, quality assurance methodologies should take into consideration the similarities and differences among spreadsheets respectively spreadsheet developers and conventional software respectively programmers while providing mechanisms of reducing errors. The challenge is to maintain a methodology in its "simplified" form (it may not be inherently simple) on the surface level which provides the user adequate control in order to check whether his/her spreadsheet is "reasonably" correct.

However, very little is known about the use of testing as a means to improve spreadsheet quality. The problem we want to address here is: How can spreadsheet programs be tested? Unlike previous testing approaches which are based on structural test adequacy criteria (i.e., based on code coverage criterion) we tackle the problem based on the functionality of spreadsheet formulas.

This work aims to devise a testing methodology for spreadsheet programs on the premise that their developers are not software professionals. The approach takes inherent characteristics of spreadsheets as well as the conceptual models of spreadsheet programmers into account and incorporates ideas from symbolic testing and interval analysis. The use of interval analysis on spreadsheets is appropriate particularly based on the observation that spreadsheets are mainly used for numerical computations. For that matter, interval arithmetic is a generalization of real number arithmetic which is the mathematical computation used behind spreadsheets.

The proposed methodology enables the user to attach intervals to the desired input and formula cells and automatically perform interval analysis to determine the conformance of spreadsheet computation, user expectation and interval computation. Moreover, the integration of the testing methodology on top of a familiar spreadsheet development system and the usability of the methodology without requiring any concept of conventional software testing techniques will facilitate the use of the approach by the large community of spreadsheet users.

The remainder of this chapter is organized as follows. The main focus of this thesis, the need for spreadsheet program testing, is briefly described in section 1.2. A highlight of the interval-based testing methodology which is based on symbolic testing and interval analysis is given in section 1.3. A description of the organization of the thesis is given in section 1.4.

1.2 The Spreadsheet Testing Problem

Spreadsheet programs are easy to write but program understanding, fault detection, and debugging are difficult. As the computational structure and the accompanying documentation are hidden from the user, maintenance is often error prone. Although spreadsheet quality suffers from different perspectives such as design and maintenance the focus of this thesis is on testing spreadsheet programs.

Formulas are very crucial elements of a spreadsheet program whose result serves as a basis for decision making. Spreadsheets are usually developed to manipulate numerical values. The manipulation is accomplished through the use of formulas. Basically there are two types of cells in a spreadsheet: input cells and formula cells. Input cells contain numbers entered by the user. Formula cells contain mathematical expressions which contain references to the values of other cells, functions, and/or arithmetic operators. Hence, it is not difficult to imagine that most of the errors will be introduced while creating formulas. In addition, some studies have already shown that the number of errors in formulas are higher than other errors in a spreadsheet program. In an experimental study of experienced users, Brown and Gould [10] found that the majority of errors were formula errors. Saariluoma and Sajaniemi [101] have identified two types of errors which are location errors and formula errors. In location errors, the formula is structurally correct but one or more of the cell references are wrong which affects the value of the formula. Similarly, Chadwick and Rajalingham [17, 93] showed in their classification and detailed description of spreadsheet errors that the observed errors were concerned with the construction and use of formulas. In addition, they stated that this result was confirmed by surveys of professional computer auditors in the United Kingdom. The main problems associated with the use of a formula as stated in [16] are choosing a wrong mathematical formula (e.g., incorrect use of SUM, average problem, etc.) to implement the required computation and the incorrect use of relative and absolute cell references for a formula to be copied to other locations. In our approach, we proposed interval-based testing to check the correctness of formulas.

1.3 Testing Approach

A program can be tested based on its functional and structural characteristics. The aim of structural testing is to measure how much a program code is exercised by a given set of test cases. To do this, different code coverage criteria are available based on the requirement of code coverage. A strong code coverage criterion requires the design of a large number of effective test cases to assure that a large portion of the code is covered during testing. This requires knowledge of the code structure and use of abstract models such as control flow graph and def-use graph to describe the code structure. On the other hand, functional testing is performed based on the functionality of the program rather than the code structure. As a result the concern is what the program is supposed to do rather than how it is written. Test cases are designed so that it is possible to observe whether the desired functionality is accordingly implemented. These techniques are targeted to professional programmers who write programs based on formal methodologies of software development.

Spreadsheet programmers who are not concerned about the formal process of software development write programs mainly on a trial and error basis. Thus, they are not expected to do testing respectively design in a formal way. However, they will benefit much and can develop quality programs if they are provided tools which do not require concepts of software engineering theory. Based on this observation and the inherent characteristics of spreadsheets, we proposed an approach which combines symbolic testing and interval analysis.

1.3.1 Symbolic Testing

Symbolic testing is a static analysis technique which requires the use of symbolic values instead of actual values for input variables in a program. The program is assumed to be executed but with symbolic values. The validity of the resulting formula (i.e., symbolic output) is determined for any arbitrary value of the input variables. If a formula is valid in such circumstances, then it is expected to be valid when actual values are substituted for the input variables. Hence, it can be said that symbolic testing lies between testing and proving program correctness.

However, unlike symbolic testing, which requires expressing formulas only in terms of input variables, interval-based testing uses intermediate variables for the purpose of narrowing down computed intervals. In addition, while symbolic testing is used to validate a formula for any possible values of the input variables, interval-based testing requires the values of the variables to be expressed as intervals and validity is determined based on the intervals provided.

1.3.2 Interval Analysis

An interval represents a range of possible values bounded by the interval's lower and upper bounds. Interval analysis is used to extend the properties of real numbers so as to provide a solution for problems involving uncertainty and approximations. Apart from mathematical problems, its application is recognized in various fields such as artificial intelligence and Systems and Control Engineering. Due to the inadequacy of closed intervals to solve a variety of problems, the classical interval arithmetic is extended to deal with open intervals and intervals containing zero values. We can see that the applications of interval analysis is mainly for applications dealing with numerical computations.

Based on the observation that spreadsheets are mainly used for numerical computations, we use interval computations on spreadsheets to detect the existence of faults. The user specifies input and expected intervals for desired input and formula cells respectively. This will be documented in a behind-the-scene spreadsheet and used to perform interval computations during the verification of a given spreadsheet. In addition, the expected intervals provided by the user are verified for reasonableness using interval analysis.

1.4 Overview of the Thesis

The main goal of this research work is to devise a testing methodology for spreadsheet programs. To achieve this goal, testing methods of conventional software were surveyed, limitations of existing quality assurance approaches were examined and an approach which takes into account inherent characteristics of spreadsheets as well as the conceptual models of spreadsheet programmers is proposed and implemented.

The remainder of this thesis is organized as follows. Chapter 2 describes the usage and characteristics of spreadsheets and the general quality issues that arise in spreadsheet programs from different perspectives such as design, testing, and maintenance. To give some background information about the types and real-life consequences of spreadsheet errors, we present some of the error incidents that are reported in the literature and a taxonomy of errors which are results of research and field audits collected by different authors. In addition, a discussion of some of the limitations spreadsheet systems have which directly or indirectly contribute to the quality problems of spreadsheet programs is presented. In chapter 3, we first present an overview of conventional software testing followed by a discussion of the similarities and differences spreadsheet programs have with reference to conventional programs. Based on these similarities and differences, an assessment of the applicability of conventional software testing methods to spreadsheet programs is made. As a result of this assessment, an approach to testing spreadsheet programs is proposed which is based on interval analysis and symbolic testing. Chapter 4 deals with the discussion of related work which are aimed at preventing and revealing the existence of faults in spreadsheet programs. We describe the approaches of spreadsheet design, formula design, visualization, and testing.

In chapter 5, we discuss the background concept of interval arithmetic which is the core concept of our testing methodology. Chapter 6 describes the interval-based testing approach which is based on interval analysis and symbolic testing. It discusses the basis for interval-based testing and defines the process of spreadsheet program testing. To detect the existence of symptoms of faults, intervals are attached to desired cells and interval computation is performed on spreadsheets. For this, a discussion of interval computation on spreadsheets is provided. In addition, a verification coverage criterion which indicates the extent of verification for a given spreadsheet is provided. Once the existence of symptoms of faults is detected, the next task is to find the location of the actual faults. A fault tracing strategy to identify the most influential faulty cells is discussed in chapter 7. In chapter 8, we discuss the design and implementation issues of interval-based testing methodology and describe its architecture. Chapter 9 concludes the thesis by providing the main ideas of the work and its contribution. A further extension of the work is also highlighted. The definitions of the basic terms used in this thesis are given in the glossary.

Chapter: 2

Problem Background

"Spreadsheets make it easy to do complex calculation-and even easier to do them incorrectly" [105].

Spreadsheets systems are widely used and highly popular end-user systems. They serve as an important basis for decisions in almost any field of a modern society. Despite their exemplary status and pioneering features, spreadsheets suffer from quality problems. In this chapter, we review spreadsheet usage (section 2.1) and the general quality issues of spreadsheet programs (section 2.2). Some error incidents and different taxonomies of spreadsheet errors are also provided. Besides the errors committed by users, the limitations of spreadsheet systems are also discussed. Finally, a summary of the main points of the chapter is given in section 2.3.

2.1 End-user Programming and Spreadsheets

In the history of computer technology, the first computers were developed for numerical computations. Due to the increasing demand of people to use computers for other applications, its use is extended in diverse applications. The development of spreadsheet systems immediately followed the development of personal computers. Spreadsheet systems have played a major role in the success and popularity of personal computers [62, 97]. The first spreadsheet system, **VisiCalc**, which was developed in 1978 by Robert Frankston & Dan Bricklin was developed exclusively for the purpose of accounting. The tabular interface was just an electronic representation of an accountant's ledger sheet. Soon, people found the tabular layout easy and appropriate to map a variety of problems directly into a spreadsheet. As a result, spreadsheet systems did not remain as a tool for accountants only; they rapidly became a tool of choice for various applications. Nowadays, they are used almost in all office environments for the purpose of numerical computation, mathematical modeling, scientific computation, tabular and graphical data presentation, data analysis and decision making, etc. The basis for all these applications is its suitability for numerical computations.

Spreadsheet programs are developed for a variety of tasks. Some are used for simple, one-time applications; others are used for complex tasks and are used frequently. For example, Sajaniemi & Pekkanen [103], in their study of spreadsheets in business and government organizations, found a spreadsheet program which contains 1092 edges of a path in the data dependency graph. While some are for personal use, others are highly important and used for organizational missions. Spreadsheet programs which are developed for organizational purposes are usually developed by a group of people with varying domain and programming experience. In their study, Nardi and Miller [74] found that spreadsheets used in the work environments are results of collaborative work by people with different levels of programming and domain expertise. Cooperative work is accomplished by successive refinement of a spreadsheet program by people of different responsibility and expertise and by exchange of templates. During the verification of a spreadsheet program, different people with different levels of expertise were also found working together.

Spreadsheet systems are widely used and highly popular end-user systems. They have actually contributed a lot to the promotion of end user computing. According to the number of users, spreadsheet systems are the most widely used next to word processing packages [108]. In a questionnaire of 373 accountants who are working in large and mid-sized firms, accountants chose spreadsheet systems over other systems (e.g., DBMS and Accounting systems) [57]. Even though this study considered only one area of spreadsheet use, it still indicates the degree of spreadsheet importance. There are different reasons for their popularity.

1. Spreadsheet systems are easy to use. To write a spreadsheet program, one needs to fill numbers to cells and the accompanying labels and then specify formulas to perform computations. For the specification of a formula, there are also a variety of built-in functions from a pre-defined library. The user needs only to identify the arguments of the functions which are mostly cell references.

2. Spreadsheet systems provide a computational power to non-programmers to do complicated sets of calculations quickly and easily [25]. Accountants, secretaries, scientists, etc. develop very important applications without the need of learning conventional programming.

3. The integration of a task-oriented formula language and tabular layout [73]. The availability of high-level task specific functions relieves the user from low-level details of programming. Spreadsheet languages provide a set of mathematical, business, and statistical functions which are commonly needed for a variety of applications. The tabular layout is a familiar and flexible display for structuring and presenting data for a wide range of applications [39]. It also serves as a guide and problem understanding framework during the problem solving process.

4. The metaphorical and visual nature of the user interface [114]. Spreadsheet systems provide immediate visual feedback about the effects of computation. The entire state of computation is visible.

Spreadsheets are an important basis for decisions in almost any field of a modern society. Besides their use in business, Filby [32] demonstrated a variety of examples used in scientific and engineering computations such as Physics and Electronic Engineering, Chemical Engineering, Molecular Biology, and Material Sciences. The ever increasing use of spreadsheets in various fields stretch them in different directions. On the other hand, there are different researches trying to extend the existing computational model of spreadsheet systems, adding to its multi-dimensionality. The spreadsheet computational model is a one-way constraint model; meaning that the direction of evaluation is from input argument values to output formula values. Stadelmann [110] and Hyvönen & Pascale [45] claim that spreadsheet systems could be best used if the computational model is extended to a two-way constraint model. The idea is to enable spreadsheets to solve for inputs from the value of a formula (i.e., back solving). In fact this is a problem of finding the inverse. The formula is supported by a constraint for which the inverse value is to be computed. For example, if a spreadsheet formula for a profit is given by profit = sales - expense, then a user might be interested to know what values of sales would result in a profit of say between 1,000,000 and 1,500,000 for a certain expense. Nardi and Miller [73] analyzed the advantages of the spreadsheet model as a general model for end-user programming.

Others investigated the use of "sheet-based" programming for other purposes. For example, Chi [22] used the spreadsheet metaphor for information visualization where the cell contains a data set (i.e., abstract data set, selection criteria, viewing specifications, etc.) instead of simple data elements and formulas; Yoder and Cohn [121, 122, 123] used the spreadsheet metaphor for designing a programming language model for concurrent computation; Hudson [42] used the computational model of spreadsheet systems for the purpose of user interface specification; Lewis [60] provided support for interactive graphics based on the basic concepts of the spreadsheet. Generally, the spreadsheet model is serving as a model for various kinds of programming environments.

2.2 Spreadsheet Quality Issue

Spreadsheet systems are used by millions of people everyday and they are the choices of many individuals and companies. The provision of computational techniques that match users' tasks makes programming easier.

Spreadsheet programs are different in some ways and these differences actually contribute partly to the spreadsheet quality problem. First and foremost, spreadsheets are developed by end-users who are not acquainted with the process of software development and as such they lack design review and control procedures that are normally associated with traditional software developed by professional programmers [57]. In spreadsheet development, the user usually starts coding before prior specification or design is accomplished and as such programming is done by trial and error. There are no distinct periods of design, coding, testing and debugging. In addition, some of the inherent properties of spreadsheet systems that seem to reduce complexity at first sight might become a burden in complex situations specifically during modification (e.g., invisibility of computational structure). The ad hoc style of spreadsheet development combined with the inherent properties of spreadsheet systems can raise quality issues. Thus, given the factual importance of spreadsheets due to the importance of the decisions based upon spreadsheet computations, spreadsheet quality needs a serious consideration.

2.2.1 Error Incidents

A number of experiments and field audits documented the existence of quality problems in spreadsheets. These studies encompass students in academia as well as professionals in real-world applications. All studies come up with different error rates but all found a significant number of errors.

One of the earliest studies of spreadsheet errors by Brown and Gould [10] found that 44% of the spreadsheets created by experienced users contained errors. A more recent and comprehensive study and collection of spreadsheet errors is the one compiled by Panko. In a development experiment, Panko and Halverson [85, 86], found that 64% - 79% of the spreadsheets created by different groups of subjects were erroneous. In addition, Panko [87] summarized the cell error rates at different stages of a spreadsheet life. He found a cell error rate of 1.9% - 5.6% in development experiments; 1.2% in field audits (i.e., in operational spreadsheets); and 34% - 55% in code inspection experiments. Note that cell error rate (CER) is a measure of the number of errors in spreadsheets which is a counterpart of faults/KLOC in conventional programming. CER is defined as the total number of errors divided by the total number of numeric cells (i.e., input numeric cells and formula cells) [81].

There are also errors that were found in operational spreadsheets which had a serious impact for the firms. For example, a U.S. contractor mis-referenced a cell in a SUM formula over a range and lost $254,000 in a bid [49]; a Dallas-based oil and gas company fired employees for decisions made based on an erroneous spreadsheet which costed the company millions of dollars [49]; UK H.M. Customs and Excise found 10% of tax payers spreadsheets containing errors in a partial analysis of the spreadsheets during 1988 - 1992. The value of the errors was reported to be more than two million pounds [14]; a financial model review by an auditing firm KPMG management consulting found at least 5 errors in 95% of the spreadsheets reviewed [94]; a Houston consultant with Price Waterhouse found 128 errors in 4 spreadsheets that had already been in use [93].

2.2.2 Taxonomy of Spreadsheet Errors

A study of errors addresses the issue from different perspectives. A commonly used strategy is to consider frequency of occurrence, real life consequences, and detail study of the actual errors themselves. In the situation where it is difficult to detect and remove all errors in a program, identification of important ones is often necessary. Beizer [7] defined a metric for bug importance which takes into account the frequency of occurrence and consequences.

Classification Schemes

A classification scheme should address the types of most frequent and important errors. In addition, the effectiveness of error prevention and detection techniques can be evaluated based on a taxonomy of errors which indicates the types, frequency and possible causes. However, as Beizer [7] indicated, there is no universally correct way to categorize faults. A given fault can be put into different categories depending on the view of the programmer and the source of the error as we can see in the following classifications. For example, typing + instead of - in a given formula may be typographical or misunderstanding of the necessary arithmetic.

Some classification schemes are available for spreadsheet errors. Panko and Halverson [87] offer a taxonomy that consists of three major categories of errors: mechanical, logic, and omission errors. Mechanical errors refer to typographical and positioning errors. Logic errors are misunderstanding of the logic of the necessary algorithm to be used in a formula. Omission errors are a result of leaving out something needed in the program. This classification is mainly based on the causes of the errors. A more general classification scheme containing Panko and Halverson's scheme is given by Rajalingham et al. [92].

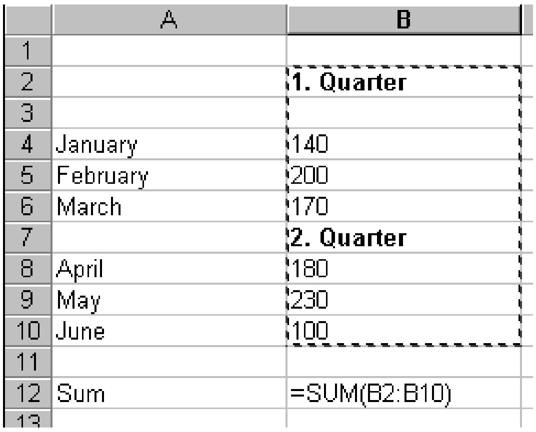
Saariluoma et al. [101], in their experimental study, categorized spreadsheet errors in two basic types: Location and Formula errors. Location errors are what are commonly termed as misreference errors. They also indicated that these errors are typical in spreadsheet programs. Formula errors contain typographical errors in formula components and what they call mathematical mistakes. Mathematical errors are a result of the inability to define the necessary mathematical expression in a formula. The main errors in this scheme are typographical, misreference, and mathematical errors.

Another classification is given by Ayalew et al. [5] from a different perspective. Instead of categorizing errors based on their cause, this scheme classifies errors by the spreadsheet concept they seem to be associated with. In this scheme there are three categories of errors: Physical area related errors, Logical area related errors, and General errors.

Category 1: Physical Area Related Errors

Errors that are typical to physical areas normally deal with missing values in the area or values of the wrong type somewhere in the area. We call this error *reference to a blank cell* and *reference to a cell with value of wrong type.* In some cases such values are entered intentionally to achieve a better structure and/or readability of the spreadsheet program. In other cases, these values result from errors.

Example 2.1: Reference to a blank/wrongly typed cell
In figure 2.1 the range for the sum spans from label **1. Quarter** *down to the last cell*



	A	B
1		
2		**1. Quarter**
3		
4	January	140
5	February	200
6	March	170
7		**2. Quarter**
8	April	180
9	May	230
10	June	100
11		
12	Sum	=SUM(B2:B10)

Figure 2.1: Reference to a blank/wrongly typed cell

of the list. The two label cells are not considered in the sum yet, but there is no hint for the user that they might influence the sum if they are changed to a number (e.g to **1** *instead of* **1. Quarter***).*

A similar error occurs when numbers are entered (or formatted) as text though the numbers are intended to be used for some calculation purposes. This is due to the fact that spreadsheet formulas take text cells and numeric cells without signaling any error message about type mismatches.

Example 2.2: Incorrectly formatted cells

Figure 2.2 shows a spreadsheet to calculate the sum of the scores of students using a SUM formula. This spreadsheet was imported from another program. The SUM function over the range B3:N3 resulted in the value 22 which is incorrect (see cell P3). The user tried to format using ***General*** *number format but the result did not change. A second attempt was to format using* ***Number*** *format but with out any decimal places (just for presentation purpose). The result did not change. The user was wondering what a miracle was going on and made the calculation by hand (see column O). While calculating manually, the user made another error (see cells O4 and O6). The numbers appear to the user as numbers but actually they are not and it is difficult for the user to see why it was wrong. It becomes visible when the cells are formatted using* ***Number*** *format with some decimal places. Those numbers which are formatted as text will not have decimal digits.*

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1																
2		Score 1	Score 2	Score 3	Score 4	Score 5	Score 6	Score 7	Score 8	Score 9	Score 10	Score 11	Score 12	Score 13	SUM	
3		**2**	**8**	**10**	**10**	**10**	**9**	**8**	**10**	**6**	**8**	**8**	**10**	**10**	**117**	22
4	Student 1	2	8	10	10	10	9	7	9	5	8	6	10	10	**95**	104
5	Student 2	0	8	10	7	6	6	8	10	3	5	6	6	9	**84**	84
6	Student 3	1	5	8	6	3	6	4	7	2	4	6	7	4	**93**	63
7	Student 4	2	8	9	10	9	6	7	7	6	5	4	10	10	**93**	93
8	Student 5	2	8	10	9	9	9	7	7	5	5	6	5	7	**89**	89
9	Student 6	0	8	10	9	8	9	8	8	4	4	4	5	4	**81**	81
10	Student 7	2	8	10	10	9	9	7	9	6	7	6	7	10	**100**	100

Figure 2.2: Incorrectly formatted cells

Another typical problem of the physical area is the impact if new values are added to the area. If the new value is inserted somewhere in the middle of the physical area, it automatically expands such that the new value and all old values are still within the area. If the new values are added by appending them to the area, the area does not expand. This leads to the error type of *incorrect physical area specification.* Generally, the incorrect physical area specification problem exists if there are cells outside the physical area which should be part of it. For the user it is not clear that those cells are not part of the physical area any more and it is common for him/her to assume that those cells influence the result of the function applied to the physical area too.

Example 2.3: Physical area specification error

In figure 2.3 the user defines a sum over an area of cells. During the life span of the spreadsheet program it turns out that more cells are needed for specifying the revenues of the salesmen. This is not a problem for extending Miller's range, but the row appended for Smith is not part of the physical area anymore. The sum cell does not yield the correct result, but the reason why the final spreadsheet instance is wrong is not obvious for the user.

A third class of typical error is the *accidental deletion of a cell* within a physical area. This leads to the already identified *reference to a blank cell* error.

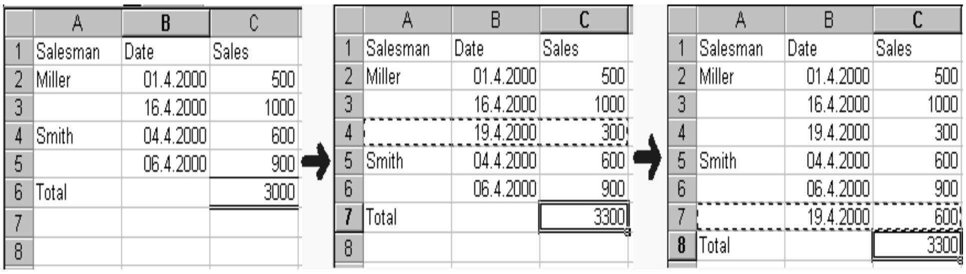


Figure 2.3: Physical area specification error

A fourth class of errors is the *physical area mixup* error. While the previous error categories are grounded on the fact that users hardly distinguish between spreadsheet programs and spreadsheet instances (input has not the distinct role as in conventional programming), this error class is due to the spreadsheet program's property which is a mixture of a problem solving tool and a presentation tool. The problem arises when two separate physical areas get mixed up. In this case, one of them cannot be defined as a physical area by the user anymore. The grouping functions have to be replaced by expressions (i.e `SUM` by multiple +). For the user it is not obvious that (s)he can specify two physical areas in two columns (see lefthand-side of figure 2.4).

Example 2.4: Physical area mixup problem

As shown in figure 2.4, the salesman spreadsheet program has to calculate a final sum over all sales and a subsum for each salesman. If the user wishes to place the final sum, the subsum and the sales in one column (i.e. for layout-reasons), the final sum has to be replaced by an expression which adds the subsums. If the subsum moves to another cell or another salesman (with a new subsum) is introduced, the user has to maintain the final sum expression. If (s)he forgets it, the final sum becomes wrong.

Category 2: Logical Area Related Errors

A logical area represents some kind of cohesion between cells. Normally a logical area originates from copying the same source multiple times and the user is not aware of the logical area where a cell belongs to.

	A	B	C	D	E	F	G	H
1	Salesman	Date	Sales			Salesman	Date	Sales
2								
3	Miller	01.4.2000	500			Miller	01.4.2000	500
4		16.4.2000	1000				16.4.2000	1000
5		18.4.2000	900				18.4.2000	900
6	Subtotal Miller			2400		Subtotal Miller		2400
7	Smith	04.4.2000	600			Smith	04.4.2000	600
8		06.4.2000	900				06.4.2000	900
9		16.4.2000	1000				16.4.2000	1000
10	Subtotal Smith			2500		Subtotal Smith		2500
11	Total			4900		Total		4900

Figure 2.4: Physical area mixup problem

A typical error is *overwriting a formula with a constant value*. This error can have many reasons like rounding errors or unexpected results of the formula. The user simply overwrites the formula result in the cell with a constant value. Of course, this value remains there even if the values in the formerly referenced cells change.

Another error that is common to logical areas is *copy misreference*. In this case, a constant value or an absolute reference is specified in a formula instead of a relative reference. This error is generally not noticed until the cell's formula is copied into another cell. If a constant cell is referenced with a relative reference, a similar problem will occur when the cell's formula is copied.

Category 3: General Errors

General errors are not explicitly associated with a physical or logical area. This category includes errors that occur in input and formula cells. An error associated with input cells is only typographical. Incorrect use of formats also affects the way a value is displayed. One might format a value as 0.2% while the intended meaning could have been 20%. This can happen to both input cells and formula cells. In addition, if a numeric value is formatted as text data, then it might affect the computed value of a formula.

Most of the errors in spreadsheets occur during formula definition [10, 101]. As stated in the glossary, a formula may involve cell references, functions, operators, and constant values. An error can occur in any of these components due to typographical errors or inability to formulate the necessary mathematical expression. These errors include cell reference errors, function errors, operator errors, boundary errors (e.g., A5 $<$ 10 instead of A5 $<=$ 10), and parenthesis errors.

2.2.3 Limitations of Spreadsheet Systems

The wide acceptance of spreadsheet systems is not only due to their simplicity but also due to their features which facilitate programming. The suppression of the low-level details of programming, the immediate visual feedback, and the availability of high-level task specific functions are commonly referred features among the others. However, in spite of their exemplary status and pioneering features, there are some limitations which directly or indirectly contribute to spreadsheet quality problem.

Different authors have indicated some of the limitations of spreadsheet systems which may influence the quality of spreadsheets [39, 80, 113]. However, the features associated with the issues are primarily designed to simplify presentation and programming, but have some negative consequences. The following are commonly raised issues.

Invisibility of Cell Information. One commonly referred drawback is the invisibility of cell information (e.g., invisibility of formula, documentation, etc.). While the values of cells are visible, other accompanying properties of a cell are not visible (except at explicit inspection). For example, the formulas which compute the values of cells are hidden. It is possible to see either the formulas or the values but not both at the same time. For a single cell, it is possible to see both but this does not give much information about the overall structure of the spreadsheet. In some cases, this locality to a single cell may help by narrowing the point of focus instead of dealing with the program as a whole, but it is also difficult to get sense of the general structure of the spreadsheet program [39]. This also indicates the difficulty to predict how changes (deleting cells, modifying cell values and formulas, etc.) to one part of the spreadsheet will affect other parts. It is often difficult to identify where the data comes from and where it goes to unless one makes a detailed examination of the relationships. This is the case for documentation as well. There is no place for program documentation and if available for a cell, again it is hidden. This is because space is optimized for result presentation. We can not make an internal documentation in the way we do it for conventional programs where each module, procedure or function is described in a readable manner (e.g., using indentations and blank lines). Hendry and Green [39] mentioned that programs are comprehended best when the whole text is available for inspection rather than an individual line at a time. Without structured documentation, it is difficult to understand how cell and range references are mapped to the problem domain interpretation. As a result spreadsheet programs turn out to be difficult to understand by other people who are not the original developers.

Spreadsheet systems (e.g., MS Excel[1]) provide a facility to annotate a cell or range. Even with this feature, sometimes it requires much text and space to describe and this might reduce the readability of the data. For example, documenting a nested conditional which involves different alternatives and understanding the logic behind it may not be an easy task. Generally, we can say that the computational structure and documentation are hidden from the user. The invisibility of such important information contributes to the inability of understanding the functionality of cells and hence leading to incorrect interpretation and computation. This also makes testing, debugging, and maintenance of spreadsheets difficult.

Cryptic Cell Addresses. Cell addresses are difficult to remember and are not descriptive enough about the data they contain. Users usually assign column and row headings for a group of cells to describe the intended meaning of the data. But in the actual formula, cell references are made through column labels and row numbers. The address is not automatically adjusted from the column and row headings. For example, the formula SUM(A1:A3) would have been easier to understand if A1 and A3 were represented in terms of their corresponding column and row headings. It is possible to assign a name for each cell but that is time consuming. In addition,

[1]All trademarks mentioned herein are the properties of their respective owners

cell addresses are used to avoid the need of variable declaration used in procedural programming.

Absence of Type Checking. Spreadsheet systems usually avoid strict type checking. A text cell and numeric cell can participate in an arithmetic formula without any symptom of type mismatch. In some cases the effect could be neutral but in other situations it could be an error (as text data is converted into a zero value). Had there been a type checking mechanism, the error in figure 2.2 would have been easily identified. Similarly, a number can be formatted to be a date type, yet this can be used for arithmetic operation with a numeric data. At the surface level, the spreadsheet system provides the user a semantic difference through formatting but below the surface they are the same data type and such errors are difficult to detect. This is one limitation that spreadsheet systems support only a few types, typically numbers, strings, and Booleans [11].

Absence of Relationships between Formulas. There is no relationship between the source and the copy of a formula. A common feature of spreadsheet languages is formula copy/paste and users rely on copying one formula to a range of cells. Based on the type of references used in the source formula, the references in the destination cells will be automatically adjusted by the underlying language. While creating a formula for copying, the user has to consider the type of references and direction of copying. The idea behind for the importance of the relationship is that if the user changes some part of the source formula (e.g., change operators or cell references), it is not automatically propagated to the copies and hence the user has to remember and find those copies and make a new copy/paste to update. The relationship between source and copy of a formula, if it exists, may help reducing the testing effort needed for the copies once the source is sufficiently tested. Since the source and copies have the same functionality and structure, incorrectness of the source in most cases (not always because incorrectness may be from referenced cells) implies incorrectness of the copies. For example, consider a formula $= A1 + B1 * C1$ defined for a cell and copied to a group of cells. Based on the values of cells A1, B1, and C1 the user may later on realize that the intended meaning was $= (\mathrm{A1} + \mathrm{B1}) * \mathrm{C1}$. After making this correction, the user has to remember where this formula was copied to and make the necessary update. On the other hand, from figure 2.2, we can see that the values of copies of a formula could be correct though the source is incorrect. The value of SUM(B3:N3) is incorrect but when this formula is copied down to calculate total score of each student, the result is correct. The other input values except those values in cells B3:D3 were entered as numbers. Similarly, correctness of the value of the source does not necessarily imply correctness of the copies (due to the effect of relative and absolute references during copying and errors in the referenced cells). Again from figure 2.2, if SUM(B3:N3) is a copy of the formula derived from the SUM(B10:N10), the value of the copy is incorrect while the source provides the correct value. However, if a relationship among source and copies is maintained, then propagation of bug fixes can be supported when the fault is in the formula itself.

Moreover, some authors mentioned the lack of development methodology to build spreadsheet programs. In conventional programming, there are widely recognized techniques for designing, testing, debugging and understanding a program. In spreadsheet programming we lack such techniques though the demand is high [26]. Some studies on users indicated the difficulties users have in testing and debugging spreadsheets [39].

2.3 Summary

As seen from experimental studies and field audits, large proportions of spreadsheets have been found to contain errors. The real life consequence of spreadsheet errors is severe to the extent of loosing millions of dollars as a result of decision made based on erroneous spreadsheets.

The first step toward a solution is the study of the type and frequency of errors. For this purpose different classification schemes have been presented.

The principal components of a spreadsheet program are formulas which are used for computations and whose result is the basis for a decision. Of course, spreadsheets are not of interest without formulas. Hence, errors occurring in formulas need a serious examination. As some studies already indicated, most of the errors occur in formula definition. Our work focuses on checking the correctness of formulas.

Besides the errors committed by users, spreadsheet systems have also some limitations which may directly or indirectly contribute to spreadsheet errors. The invisibility of computational structure and the accompanying documentation, the obscure meaning of cell addresses about the data they contain, the lack of strict type checking, and absence of relationships between source and copies of a formula are commonly encountered issues.

Chapter: 3

Assessment of Traditional Testing Methods

This chapter reviews the testing techniques in the imperative paradigm and discusses their applicability in testing spreadsheets. Section 3.1 presents an overview of software testing techniques. A reader who is familiar with the testing theory in the imperative paradigm may skip this section. Section 3.2 discusses the characteristics of spreadsheet programs. An analysis of the similarities and differences between spreadsheet programs and procedural programs is presented in section 3.3. Out of the analysis, the implication for a testing methodology is discussed in section 3.4. Section 3.5 concludes the chapter by giving insight about the testing approach which seems appropriate for spreadsheets.

3.1 An Overview of Software Testing

Software is developed to solve a variety of problems be it simple, moderate or complex. All sizes of software require a systematic and effective method of testing to provide the desired functionality and quality. Software testing is a very crucial activity in the development of software systems though it is an expensive and labor-intensive task. Estimates indicate that software testing accounts for up to 50% of the cost of development [7, 71, 109], and even more in safety-critical systems. One of the factors for the expensiveness of software testing is the difficulty of automating it. Even if it is automated, it is still in general impossible to develop 100% fault-free software. This is due to the infiniteness of the input size to be used for testing, the large number of possible paths to be executed and other external parameters. As a result, criteria for selecting input representatives and crucial paths are used.

In the literature one can see that there is no unique definition of software testing. Its definition is stated from different perspectives by different people. It is commonly considered as techniques of checking software by executing it with data. Myers [71] defined it as *"the process of executing a program with the intent of finding errors"*. Hetzel [40] defined it as the process of establishing confidence that a program or system does what it is supposed to do. The two definitions given by Myers and Hetzel represent defect testing and validation testing respectively.

Defect testing focuses on the exposure of hidden faults. Test cases are designed to cause the system to function incorrectly as opposed to validation testing where test cases are designed to show the system performs correctly. The main focus of defect testing is on demonstrating the presence rather than the absence of program faults and it will be successful if it reveals a fault in the program.

Validation testing is concerned with showing that the software works. This approach is a positive (constructive) process and is less likely to uncover faults as most people have a tendency to show that their program really works. The need for testing is that after development we want to be sure that the product is fault free. Actually it is hard, if not impossible, to guarantee that the system developed is completely free of faults. The main objective of testing is to detect the presence of faults. Testing cannot demonstrate the absence of faults. If testing is performed with the objective of finding faults in the software, then it also demonstrates that the software functions appear to be working according to the specification [7].

The two different definitions indicate a difference in the objective of testing. The objective of defect testing is to find faults in a software while the objective of validation testing is to demonstrate that there are no faults in the software. Beizer [7] has indicated the limits of both objectives as follows. If the objective of testing is to detect bugs, then when are we going to stop testing? On the other hand, if the objective of testing is that the software works, then even an infinite number of tests will not prove that it works. Hence, he concludes that testing is targeted for risk reduction.

There are different testing methods used in the imperative paradigm [7, 71, 88]. These methods are usually divided into two main categories: static testing and dynamic testing. Static testing is analyzing the program without executing it with data as opposed to dynamic testing which executes the program with data.

3.1.1 Static Testing

Static testing techniques do not require the execution of a program with data. Rather, they focus on investigating the source code in its static form looking for possible faults. Assessment of the quality of a program is undertaken irrespective of its run-time behavior. Static techniques, if successful, enable us to detect faults directly unlike dynamic testing which provides only symptoms about the existence of faults. Static testing techniques can be used in different forms such as program proving, symbolic execution, anomaly analysis, inspections, and code-walkthroughs [24, 50].

Program proving is to establish a mathematical proof about program correctness. It focuses on showing the equivalence between a program and its specification in the form of a mathematical proof. It uses the method of inductive assertions and requires one to write assertions about the program's input conditions and correct results. The program is proved to be fault free if the assertions can be proved mathematically.

Symbolic execution sometimes referred to as symbolic evaluation or symbolic testing is the execution of the program with symbolic data [20, 41]. The execution is not in the traditional sense but the program is assumed to be executed with the symbolic inputs. The inputs to the program are not numbers but symbols representing the input data, which can take different values. The outputs are symbolic formulas of input values. The formulas can be checked to see if the program will behave as expected. At the end of the symbolic execution of a path, the output variable will be represented by expressions in terms of symbolic values of input variables and constants. The output expressions will be subject to constraints (conditions). A path condition at a statement gives the conditions the inputs must satisfy for an execution to follow the path so that the statement will be executed.

Anomaly analysis is concerned with the identification of anomalies that cannot be detected by the programming language. Anomaly analysis techniques are used to identify existence of unexecutable code in a program, redefinitions of variables without uses, uninitialized variables, and initialized but never used variables [24]. Dataflow analysis can also be used to identify some of the anomalies in a program. Anomalies do not necessarily indicate the existence of faults but they are hints for potential faults.

Inspections and walk-throughs are informal techniques. Inspections are carried out by programmers reviewing design or code with an intent of finding faults. A code walk-through is an analysis of code as a cooperative, organized activity by several programmers. Programmers may use test cases and simulate execution of code by hand. The difference between code inspection and code walk-throughs is that inspection is targeted explicitly at the discovery of commonly made errors.

3.1.2 Dynamic Testing

Dynamic testing involves the execution of a program using test data similar to the real data that will be processed by the program. Unexpected outputs can be used to infer the existence of faults. The basic procedure of dynamic testing is to provide the program with inputs, executing the program, and then compare the output of the program with the expected results. Figure 3.1 depicts the overall procedure of dynamic testing [91]. Since more information can be obtained during the execution of the program, there are a variety of testing methods that belong to dynamic testing. The two common dynamic testing methods are functional testing (also called

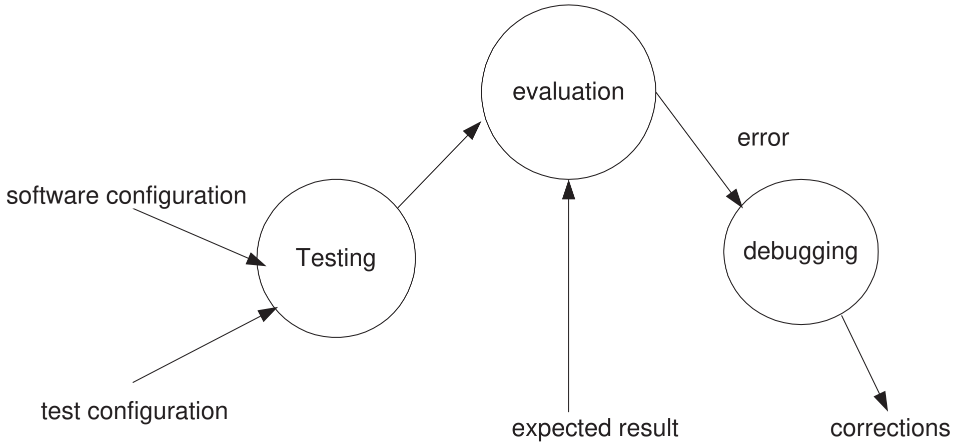


Figure 3.1: Test Information Flow [91]

black-box testing) and structural testing (also called white-box testing). Structural testing is concerned with the implementation of the program while functional testing is concerned with the function that the program is supposed to perform. Since the two methods can uncover different types of bugs, they are complementary and used in combination to design and construct a comprehensive test data set.

Structural Testing

Structural testing involves the execution of the program with test cases derived from the structure or detailed design of the program. Test cases are designed based on the internal control flow structure or data dependency. Another structural testing method is mutation testing which is based on injecting known bugs to programs. The structural testing method is mostly used in unit testing. Test cases are systematically chosen inputs which have the highest likelihood of revealing faults in the program.

1. Control Flow Based Testing

Control flow based testing uses the program's control flow as a structural model for generating test cases [7, 50]. It requires testing the logical paths of the program by providing test cases which exercise specific sets of conditions and/or loops. There are different control flow based testing techniques which define criteria for selecting a set of test paths through the program. The criteria are used to measure the thoroughness of the test and also provide a way to determine whether testing is complete. The basic control flow based testing techniques are statement testing, basis path testing, branch testing and paths testing.

Statement testing requires the design of test cases to execute each statement of a program at least once. In terms of the control flow graph of a program, statement testing requires the execution of all nodes in the graph. It is the weakest of all contrl flow based testing techniques.

Basis path testing uses McCabe's program complexity measure (called cyclomatic complexity) and the mathematical analysis of control flow graphs to guide the testing process [69, 119]. Cyclomatic complexity represents the number of independent paths in the basis set of the unit. If the cyclomatic complexity of a unit is n, then there are n distinct paths and hence test cases should be designed to execute the n distinct paths. The cyclomatic complexity (M) can be computed in any one of the following ways.

1. $M = E - N + 2$ where E represents the number of edges of the flow graph and N number of nodes of the flow graph.
2. M = number of regions of the flow graph.
3. $M = P + 1$ where P is the number of predicate nodes contained in the flow graph.

The principle is to test a basis set of paths through the control flow graph of each unit. This means that any additional path through the unit's control flow graph can be expressed as a linear combination of paths that have been tested.

Branch testing is performed with the objective of traversing each edge of the control flow graph at least once during testing. It requires the execution of tests so that each decision takes on all possible outcomes at least once. We can easily see that branch coverage subsumes statement coverage. In other words, a set of test cases satisfying branch coverage will also satisfy statement coverage. The drawback of branch testing is that it can evaluate to true and false without exercising all the conditions if the decision involves compound conditions. To alleviate this problem, test cases should be designed so that all sub-conditions evaluate to true and false. Since the combination of values of all conditions may not result in the evaluation of the decision to true and false, the combined requirement decision/condition coverage is imposed. This requires all the decisions and all the conditions in the decisions take both true and false values. In addition, each component of the condition should be tested to detect possible types of faults in the condition. A compound condition includes Boolean operator, Boolean variable, relational operator and arithmetic expression. If a compound condition is incorrect, then at least one of the components is incorrect. There are a variety of condition testing strategies corresponding to different types of condition usage in a program. For example, Tai [111] has proposed two testing strategies namely BOR (Boolean Operator testing) and BRO (Boolean and Relational Operator testing) for compound conditions. The BOR testing requires a set of tests to detect faults in expressions involving Boolean operators. Similarly, the BRO testing requires a set of tests to detect faults in expressions involving Boolean and relational operators.

Path testing requires the execution of all possible paths in the control flow graph at least once. Path testing can reveal faults which may not be detected by branch testing. Test cases designed based on path testing force the execution of all possible paths in the program. However, this technique is less practical as the number of possible paths grows rapidly even for small programs which involve loops. In addition, some paths may not be reachable and the identification of such paths is also difficult. Since the number of possible paths in a program could be very large and path testing cannot reveal all types of faults in a program, other alternatives have been investigated to fill the gap between branch coverage and path coverage [7, 35].

2. Dataflow Based Testing

Dataflow testing is based on the def-use graph which is constructed from the control flow graph of a program. The basic idea behind dataflow based testing is to make sure that the definitions of variables and their subsequent uses are exercised by some test cases. It requires the selection of test data that exercise certain paths from a point in a program where a variable is defined to points at which the variable definition is subsequently used. Variable occurrences in a program can be **def**, **c-use**, and **p-use**. The definitions are given in [50] as follows:

Def represents the definition of a variable where it is given a new value. C-use represents the computational use of a variable where the variable is used in an assignment statement, read/write statement, etc. P-use represents the use of a variable in a predicate statement.

For example, in the assignment statement $x = y + 5$, x is defined (i.e., def) and y is used for computation (i.e., c-use).

There are different test criteria defined to measure the coverage of the definitions and uses of variables. Rapps and Weyuker [95] have proposed a family of test case selection criteria based on the types of occurrences of variables in a program. Ntafos [75] and Laski and Korel [59] have also proposed test adequacy criteria based on dataflow as alternative to the control flow based measure of test adequacy. The basic criteria are all-defs, all c-uses, all p-uses, all-uses, and all-du-paths. The definitions of the basic dataflow testing criteria are given below [7, 50].

1. The all-defs criterion requires that for each variable definition, the testing paths should cover at least one sub-path from the definition to a use (c-use or p-use) of the definition.

2. The all c-uses criterion requires that all computational uses should be exercised by some test.

3. The all p-uses criterion requires that all predicate uses should be exercised by some test.

4. The all-uses criterion requires that all uses (c-use and p-use) should be exercised

during testing. This criterion requires at least one sub-path from each variable definition to every c-use and every p-use of the definition be included in the test.

5. The all du-paths criterion requires that every simple sub-path from each variable definition to every c-use and every p-use be exercised by some test.

In some cases a combination of c-use and p-use can be used to achieve a better coverage than the individual criteria. These combinations are all c-uses/some p-uses and all p-uses/some c-uses. The main task in dataflow testing is the determination of possible dataflow paths in the program (which requires static program analysis) and the recording of the paths executed during testing (dynamic analysis) [35]. Comparing with control flow based testing techniques, dataflow based testing is weaker than all paths testing but stronger than branch and statement testing. A general hierarchy of subsumption for all control flow based and dataflow based testing criteria is given in [7]. In addition, there are different experimental studies carried out to measure and compare the effectiveness of different testing criteria. For example, Hutchins et al. [43] have experimentally demonstrated that control flow and dataflow criteria are complementary in their effectiveness.

The computation of def-use paths is accomplished with the aid of slicing techniques. A program slice consists of the parts of a program that affect the values computed at a certain point [112]. Usually, program slices are computed based on a slicing criterion which contains a line number and variable. Program slices can be computed either statically or dynamically. A static program slice with respect to a given slicing criterion contains a set of statements that could potentially affect the value of the variable at the given position. On the other hand, a dynamic slice contains only those statements executed based on a given input data. We can see that a dynamic slice is a subset of a static slice of a program with respect to a given slicing criterion. Slices can also be computed in forward or backward direction. A forward slicing computes those statements which will be affected by a given variable whereas backward slicing computes those statements which will affect a given variable. The above dataflow testing criteria use slicing techniques to identify the def-use paths to be included in testing. This is accomplished by walking on the def-use graph which is constructed out of the control flow graph representation of the program. Besides their use in dataflow testing, slicing techniques are also used in debugging and maintenance. During debugging, dicing is used to guide the fault localization procedure to the most likely faulty statement by removing those statements which appear to be correct from the dynamic backward slice.

3. Mutation Testing

Mutation testing is a fault based testing technique. Instead of designing test cases to execute certain paths in a program, mutation testing requires the injection of known faults (simple syntactic changes) into a program [50, 117]. Then test cases are designed to detect the seeded faults. The effectiveness of the test data set is measured by the percentage of mutants killed. Different versions of the program (mutants) are generated by introducing faults using mutation operators. If a test case is able to generate different values for the original program and the corresponding mutant, the test case is successful. Mutation testing is based on two assumptions: the competent programmer hypothesis and the coupling effect [117]. The competent programmer hypothesis assumes that competent programmers tend to write nearly "correct" programs. That is, programs written by experienced programmers may not be correct, but they will differ from the correct version by some relatively simple faults such as off-by-one fault. The coupling effect states that a set of test data that can uncover all simple faults in a program is also capable of detecting more complex faults. Even though the basic goal of testing is detecting faults in the original program, mutation testing is targeted first at generating effective sets of tests.

As mutation testing is performed from a different perspective, it is difficult to make direct comparison with other control flow based and dataflow based testing criteria. However, there are some experimental studies showing its relative strength by comparing it with dataflow testing criteria. For example, Offutt et al. [79] carried out an experimental study to compare mutation testing and all-uses dataflow testing. Their result indicates that mutation testing offers more coverage than the all-uses dataflow but at a higher cost. The major problem with mutation testing is that the number of mutants could be very large which limits its application to large programs. Some methods (e.g., selective mutation) have been suggested to reduce the computational expenses of this testing technique [76, 117].

Functional Testing

Functional testing is performed based on the functional requirements of the software and its main focus is the functionality and features of the system. It considers the user's point of view of the system and requires the derivation of sets of input conditions that will fully exercise all functional requirements for a program. Functional testing particularly helps to identify input classes, boundaries of data classes and combinations of input data to design appropriate tests [69]. This method is complementary to structural testing as it uncovers a different class of bugs. Commonly used functional testing techniques are equivalence partitioning, boundary value analysis, and cause-effect graphing.

1. Equivalence Partitioning

As the name indicates, equivalence partitioning involves the division of the input domain of a program into classes of input conditions where each class is a representative of a large set of other possible tests. An equivalence class represents a set of valid and invalid states for input conditions. A test case is designed to uncover classes of errors in an equivalence class. Its main focus is to minimize the number of test cases needed to cover the input conditions.

2. Boundary Value Analysis

Boundary value analysis can be used to derive test cases based on the equivalence classes. The boundary values of the equivalence classes are used to derive test cases with the intent that boundaries are likely to be the causes for bugs. In addition, output conditions are also explored by defining output equivalence classes. This approach attempts to identify boundary conditions for each equivalence class. The conditions are then used to create test cases containing input values that are on, above, and below the edges of each equivalence class.

3. Cause-Effect Graphing

Cause-effect graphing is a technique that provides a concise representation of logical conditions and corresponding actions. It helps to select test cases that explore combinations of input conditions. It is a Boolean graph describing the semantic content of a written functional specification as logical relationships between causes (inputs) and effects (outputs). This approach involves identifying specific causes and effects which are outlined in the requirements document. Causes are conditions which exist in the system and which account for specific system behaviors known as effects. Effects can be system outputs that are end results of the processing. The causes and effects are transformed into a cause-effect diagram that can be used to create test cases. Cause-effect graphing helps to clearly identify every function's inputs and expected results.

Generating Test Data and Oracle

One of the most challenging task in software testing is the actual generation of test cases. Test data can be derived from both the specification and the program's implementation which have the highest likelihood of revealing faults. Test data are program inputs which satisfy a given testing criteria. The generation of test data is based on a given criterion where the criterion provides a mechanism of ensuring the completeness of tests. For example, branch coverage requires the generation of test cases which exercise the true and false branches of every decision. Most automated test data generation approaches focus on deriving tests from the program's source code and they are mainly targeted to unit-level testing. These are structural-oriented test data generators which attempt to cover certain structural elements in the program (e.g., statement coverage, branch coverage, dataflow coverage, etc.) [28, 31, 54, 78]. On the other hand test data generators based on data specification generate test data from a formal description of the input domain [64]. Automated test data generation approaches which derive tests from the program's specification require the description of the specification in some formal way [77]. Random test data generators [118] create test data according to some distribution of the inputs without satisfying any test criterion.

Testing is commonly carried out with the assumption that there is some mechanism which will determine whether or not the result of test execution is correct. In order to infer the existence of faults in a program, the result of the program needs to be compared with the expected result of the program. The preparation of the expected result could be done either manually or with the help of a test oracle. "A test oracle is any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a tested object" [7]. An oracle is thus a mechanism that provides the correct behavior of the program for the test cases. An ideal oracle is an automated one which always provides a correct result for each test case. However, fully automated oracles are difficult to create, and human oracles are commonly used for this purpose. The specification is the main source to determine the correct behavior of the program. However, during modification of an existing software, regression test suite could also be a possible source. When a formal specification exists, an oracle can be generated automatically from the specification [89, 90]. Generally, an oracle constitutes two components: oracle information (what constitutes correct behavior) and oracle procedures (which check the test results against the oracle information).

Regression Testing

While the previous approaches (see section 3.1.2) are targeted to testing during development, testing of evolving software or testing during maintenance requires a different approach. One of the major tasks during maintenance is the re-testing of the modified software. Re-testing the software with all the original test cases is very expensive. Hence the goal of re-testing (i.e., regression testing) is to re-test the new program based on the modifications made while maintaining the same testing coverage as completely re-testing the program [34, 37]. This involves the issue of selecting old test cases to be rerun and designing additional test cases corresponding to the change. Therefore, regression testing attempts to validate modified software and ensures that no new faults are introduced into previously tested code [37, 98]. Though regression testing is used to test the changed part of the program, it should also be used to test the part of the program which is affected by the change. Thus, the first step will be to identify the changed part and then the part of the program which is affected by the change. This is usually accomplished using dataflow analysis [34].

3.2 Spreadsheets and Spreadsheet Programmers

Spreadsheet programs are software developed by end users. The detailed work of programming is automatically performed by the spreadsheet system itself. Input and output are automatically performed. For example, to enter input into a spreadsheet, one simply enters a value into a cell - no separate input statement is required. In contrast, a procedural program must be explicitly programmed to accept input from the user and to display the output. This easy and declarative way of programming has an effect on the psychology of the user. Users do not consider themselves as programmers and may not even consider their work as programming and as a result do not make the necessary design review and control procedures to improve the quality of their spreadsheets. However, they are programmers whose computational task is in a certain domain. They are programming to solve a given problem which involves expressing a computation through a programming language. The only difference to spreadsheet programming is that it provides a highly interactive, declarative environment and as a result program development is easy and fast.

Of course, spreadsheet programmers are not professional programmers; they are end-users who are not familiar with the process of software development. Nardi and Miller [73] describe this issue as follows: "Spreadsheet developers are not programmers; they are business professional or scientists or other kinds of domain specialists whose jobs involve computational tasks". They prefer spreadsheets as their best tool because no formal training on designing and programming is required and they can develop working applications in a short period of time [26]. Their model of the problem is directly mapped to the tabular layout of numbers and text. In contrast professional programmers write a program based on a design which in turn depends on some conceptual model or specification.

3.2.1 Spreadsheet Languages

The emergence of end user programming facilitated the ease of programming for those whose work requires only some computational tasks. Among the variety of tools used in end user programming, spreadsheet languages are the most common one. Spreadsheet languages belong to domain specific and visual programming languages.

Domain Specific Language

Spreadsheet languages can be considered as domain specific languages since they are designed to support problem solving in a particular problem domain. They support mainly problems which fit into a grid representation (i.e., table-oriented computational problems) [124]. Hendry [38], Casmir [15] and Ambler [3] demonstrated through a variety of examples (e.g., the towers of Hanoi and sum of arbitrary number of values) that spreadsheet languages loose their power as the generality of the problem to be solved goes far away from domain specific tasks. A domain specific language is a small, usually declarative language that offers expressive power focused on a particular problem domain. In contrast general purpose programming languages are composed of a fairly large number of primitive functions and constructs. Domain specific languages such as spreadsheet languages offer a small number of primitives that map directly onto operations that users within a specific range of applications need. These small number of primitives are high-level task specific functions which map to tasks in the domain the user understands [72]. In their study, Nardi and Miller [72, 73], found that users develop working applications using only a few number of functions. Spreadsheet languages offer an expressive power which enables to express domain knowledge that shows relationships between entities in the domain itself. Programming requires mapping of the problem directly to the two dimensional grid structure.

Visual Programming Language (VPL)

Visual programming languages have got a variety of definitions: from languages which provide GUI (such as Visual Basic) to languages which use visual representations to accomplish programming. This variation is based on the extent of visual expression provided in the language. The extent of visual expression (i.e., meaningful visual representations used as language components to achieve the purpose of programming) is used in [107] as one dimension to classify visual programming languages.

A classification given by Shu [107] categorizes visual languages into three:

1. Languages designed for the handling of visual information. These are languages used for the processing of visual information. For example, a textual language used to manipulate a picture.

2. Languages designed to support visual interaction. These are languages which are used to represent an object visually. For example, a textual language can be used to describe the appearance of an object using the properties POSITION, COLOR, SCALE, etc.

3. Languages which use some visual representations to accomplish the task of programming. For example, in a spreadsheet, cells (in essence variables) are visually represented to accomplish the necessary programming task. In spreadsheet programming, the geometrical layout of data plays a major role. Generally, non-procedural programming languages which use tables or forms as visual expressions fall into this category.

This classification encompasses the definitions given by Chang [19] and Ambler [3]. Ambler considers as visual programming languages only those defined in item 3 above. Another definition given by Myers [70] considers the dimension of the program specification. According to Myers: "A visual programming language is any programming language that allows the user to specify a program in a two-(or more)-dimensional way. A VPL allows programming with visual expressions - spatial arrangements of textual and graphical symbols." This definition clearly avoids the confusion of considering languages such as Visual Basic as visual programming languages.

The goal of making programming and program understanding easier has been achieved through simplicity, concreteness, explicitness, and responsiveness [12]. These characteristics are provided to

- reduce the number of concepts needed to program (e.g., number of variables)
- allow data objects to be explored directly (i.e., the programmer can see, explore, and change specific data values)
- explicitly depict relationships (e.g., dataflow diagrams)
- give immediate visual feedback of updated computations during editing

Green [33] indicated that spreadsheet languages contain many of these features of VPLs - dataflow, aggregate operators, and a visual formalism even though they do not contain explicit graphical lines showing dataflow between cells.

Nardi [72] considers a spreadsheet language as a hybrid visual programming system which combines text and graphics. The graphics part which is the grid structure is used to make the programs state visible (showing current values of cells in the table). The textual part provides a compact formalism for writing program instructions.

From the definitions and characteristics of visual programming languages given above, we can see that spreadsheet languages are VPLs to a large extent. This particularly holds from the perspective that visual programming is used to describe any system that lets users specify a program using a two dimensional notation. In such a system, the interaction of the user is with a two dimensional representation.

On the other hand, the structure of procedural programming languages are based on one-dimensional, textual (i.e., statement by statement) representations and hence are not visual programming languages. The linearity mimics its internal representations.

3.2.2 Spreadsheet Programming

The development of a spreadsheet program is a form of programming which is carried out by end-users. It consists of understanding the problem, devising a plan to achieve the set of goals, and implementing the proposed plans [104].

However, writing a spreadsheet program does not require a course of conventional programming. The user needs only to structure the problem in a tabular layout and specify relationships among cells through the use of formulas. The main problem solving activity is based on the use of formulas for the computation. A formula is a way of using the values stored in other cells for the intended computation. Placing those values into cells does not require a variable name. The column letters and row numbers replace the task of declaring variables. Cells are named by their position in the grid. The details of conventional programming such as data type declaration are not needed. Types are determined dynamically as the user enters data to a cell (formats also affect the type of data). Copying a formula across a group of cells is a common activity. Formula copies and cell references replace iterations and assignment statements respectively. The availability of high-level task-oriented functions eliminate the need of algorithm design. Even though users do not realize it, they are actually using program constructs such as loops and assignments. However, such concepts are hidden from the user. Nardi and Miller [73, 74] discussed the characteristics spreadsheet languages provide for end-user programming. Among them is the property that spreadsheets shield users from low-level details of traditional programming. They allow users to think in terms of tabular layouts of adequately arranged and textually designated numbers.

Ambler [3] summarizes spreadsheet programming as follows:

- The programming environment is interpretive and feedback is immediate.
- No notion of variables.
- Cells do not change values in the course of an execution (except cyclic dependencies).

- No type declaration since no notion of variables.

- The order of evaluation is derived, rather than specified. No concern of control flow as control flow is entirely derived based upon computational dependencies.

- No concern of input/output.

3.3 Comparison of Conventional and Spreadsheet Programs

The spreadsheet paradigm is gaining increasing importance not only in developing ordinary spreadsheet programs but also for other purposes too (see section 2.1). However, there are no widely accepted design, testing and debugging methods in this paradigm. Devising a testing methodology for spreadsheets requires the investigation of the suitability of existing testing methods to this new paradigm until it possesses its own well defined methods. The imperative paradigm is the basis for devising methods of designing, testing, debugging, etc. software in the other paradigms. Hence, the main issue is what are the differences and what is the impact of these differences in testing spreadsheet programs. For example, to test an object oriented program, one cannot directly apply traditional testing methods. While traditional software is based on procedures, object-oriented software is based on objects which contain procedures and data together. The testing methods for object-oriented programs are adapted based on investigations by considering the impacts of encapsulation, information hiding, and inheritance [6, 23, 58].

In order to investigate how spreadsheet programs will be tested, it is essential to consider the similarities and differences between conventional and spreadsheet software. Rothermel et al. [100] have identified three classes of differences between spreadsheet and procedural programs.

1. **Order of evaluation**. While the order of evaluation of spreadsheet programs is data dependency driven (i.e., based on data dependency that exist between cells), order of evaluation of procedural programs is control flow based.

2. **Interactivity**. Spreadsheets are interactive which means that they provide immediate visual feedback about the effects of changes (e.g., automatic recalculation of all dependent cells when a source cell is modified). Procedural programs are noninteractive.

3. **Expertise of users**. Users of spreadsheet systems are mainly non-professional programmers who are not expected to know the formal process of software development. On the other hand, developers of procedural programs are professional programmers who have the necessary training of software development.

In addition, other differences can be observed from different perspectives.

1. **Structure of code**. The structure of a spreadsheet program is two dimensional while the structure of a procedural program code is represented linearly. In spreadsheet programming the geometrical layout plays a major role. The placement of code is guided by the tabular layout of the results.

2. **Separation of input/program/output**. There is no explicit separation of input, program, and output from the user's point of view. The cell contents of a spreadsheet contain both the input and the program while the visible part of the spreadsheet is the output. The main part of the spreadsheet program (i.e., formula) is hidden below the surface which indicates that the computational structure is not readily available to the user. In procedural programs, there is a clear separation between input, program, and output.

3. **Conceptual view of a program**. Conceptual view of a spreadsheet program is based on data dependency relations while the conceptual view of a procedural program is based on control flow. A conditional in a spreadsheet program is easy to understand because it does not transfer control from one part of the spreadsheet to another. Its effects are local to the individual cell [72]. Since cells use other cells' values to calculate their own values, a data dependency is established in the spreadsheet. From this perspective, spreadsheet programs can be considered as dataflow driven.

4. **Declarativeness**. The ease of use of spreadsheet languages is also derived from the fact that they are declarative [67]. The value of a cell is computed by

the formula associated with the cell. The detailed procedure of computation is transferred to the language and hence is no longer the programmer's responsibility. Formulas describe relations which specify what is to be computed. Users' understanding of the dependencies is at a higher semantic level. Therefore, users will most likely check the correctness of formulas based on their functionality rather than their internal structure.

3.3.1 How Suitable are Traditional Testing Methods for Spreadsheets?

Conventional software testing techniques are based on specifications and implementation. Specification-based testing techniques require a specification where the behavior of the software is described. Code-based testing techniques require knowledge of the internal structure of the code and they are mainly based on control flow and dataflow.

In spreadsheet programming, neither a specification is available nor a spreadsheet programmer has the expertise to design and execute effective test cases. With the absence of a specification, specification-based testing techniques are not suitable. In addition, absence of a specification implies that an automated test oracle is not imaginable. Even with the presence of a specification, generating an automated test oracle is a difficult task. Besides this, fully automated test oracle is very expensive since it is a complete alternate implementation of the software under test which provides the expected outcome for each test case [7, 65]. In such situations, the tester (the user in spreadsheets case) plays the role of a test oracle and provides the expected behavior during testing.

The concept of control flow through a conventional program does not map readily to a spreadsheet. Explicit flow of control may exist only within a cell when the cell's formula definition contains a condition. In spreadsheets, control flow is derived from the data dependency relations. The conceptual view of a spreadsheet program is based around data dependencies rather than control flow. Therefore, it seems rational to investigate a spreadsheet based on data dependency conceptualization. However, the dataflow structure is hidden and is not easily accessible to a user. Of course, there are tools such as MS Excel's arrow tool which provide a visual representation of cell dependencies using arrows. For a detailed discussion of arrow tools and other visualization tools see section 4.2.1 of chapter 4. In spite of that, it is unlikely that a user will design test cases based on def-use paths. Both control flow based and dataflow based testing focus on selecting paths to execute during testing. Hence, test case design based on control flow and dataflow paths is not feasible for spreadsheet users.

Static testing techniques such as anomaly analysis, inspections and walk-throughs can be used for any type of software but are not sufficient by themselves. They rather provide additional support to formal testing techniques. The faults that can be revealed using these techniques do not overlap with the faults that can be revealed by the formal testing techniques. For example, dataflow analysis helps to identify anomalies such as cells which have numeric values but are never used in computations. Such cells are not errors by themselves but they are likely to be so and could be indicators for further examination. Dataflow analysis doesn't help us to check the correctness of computations and hence its application is limited only to serve as an additional support.

Program proving which is mainly used for safety critical systems is not at all evident in spreadsheets. Let alone in spreadsheets, its use even in conventional software is limited to safety critical systems since it is an expensive technique (in terms of human expertise requirement and development cost).

On the other hand, symbolic execution seems applicable to a certain extent. The main objective of symbolic execution is to express a formula in terms of its input variables. The formula can be checked to see if the program can behave as expected. In the case of spreadsheets, it will be helpful if the input cells are described by a range name instead of simple cell address. For example, it is easier to understand if a net income formula cell says **gross - tax** instead of something like **B15-B16**. Symbolic execution enables us to see which input cells are involved in the computation of a formula and may help to identify those cells which are not included in the formula. However, expressing formulas only in terms of input cells may not give the intended meaning and could be difficult to judge its validity. For example, a formula may be better understood if it is expressed in terms of intermediate cells instead of input cells. Intermediate cells are used to break down a formula in some meaningful way. Symbolic execution is also used to generate test cases by expressing a formula in terms of input variables.

3.4 Implications for a Spreadsheet Testing Methodology

At a unit level testing, correctness of a unit implies correctness of behavior and correctness of implementation. The expected outcome is normally derived from the specification or from a test oracle. In the absence of specifications, programmers provide such behavior during testing. Correctness of implementation requires detailed knowledge of the internal structure and the design of appropriate test cases based on a certain coverage criterion. This applies rather to professional programmers who develop programs which will be used by different users with different data.

The root of spreadsheet programming lie in the definition of formulas. Formulas are the basic units of a spreadsheet program. Users define formulas without any concern of its detailed execution. In spreadsheet programs, users want to make sure that their spreadsheet formulas are correct with respect to the actual data they are working on. For a formula, correctness corresponds to the equality of computed value of the formula with a pre-calculated value. However, pre-calculated values may not always be available. Hence, the reasonableness of the computed value is used to judge the validity of the formula. Users usually have a gut feeling of the range of reasonable values for each given cell though.

In order to test based on the internal structure of spreadsheet code with the objective of covering some part of the code during testing (i.e., code coverage-based testing), the user must first dig out the hidden data dependency relations. This corresponds to constructing a dataflow graph to select test cases for a procedural program. However, the problem which is frequently mentioned in spreadsheet programs is that the computational structure is hidden and dependencies are hard to follow [5, 21, 26, 102, 106]. With this hidden implementation structure, it is difficult for a user to understand and hence design test cases for a certain coverage criterion. Hence, it does not seem feasible to expect a user of a spreadsheet to have such an abstract model to design test cases.

The other possibility could be to apply random testing. This involves selecting a random test data without the intention of satisfying any coverage criterion. However, this alone does not give any valid prediction of the program's reliability. Besides its inability of assuring coverage, it may be difficult to predict the desired outputs of random test data.

Out of this analysis, we can see that spreadsheets are software too and approaches are needed to improve their quality. However, an approach to address spreadsheet quality problems should be appropriate for end users. Therefore, to be successful, approaches should take into consideration that rest on the conceptual models users have instead of on concepts targeted to professional programmers. In other words, user-centered approaches will have a better chance to be practiced by the large community of spreadsheet users.

3.5 Summary

Spreadsheets are software developed by end users using a certain domain specific language. At first sight, this might give the impression to consider conventional quality assurance techniques for spreadsheets. Conventional quality assurance techniques, in their very nature, are targeted to professional programmers who write specifications and validate their products based on specifications and some code coverage criteria. As a support of the formal verification techniques, other quality assurance techniques such as inspection, reviews, and dataflow analysis can be used at various stages of software development. In spreadsheet programming where program development is non-algorithmic and largely trial and error based, we don't expect the user to do verification based on conventional quality assurance techniques. However, inspections and reviews can be used to support other formal quality assurance techniques, but they are by no means sufficient by themselves.

To be successful, approaches should take into consideration that rest on the conceptual models users have instead of on concepts targeted to professional programmers. Hence our approach doesn't depart from conventional testing wisdom. We rather propose an approach based on the purpose spreadsheets are used for and the expertise of users. Spreadsheets are mainly used for numerical computations by end users. Hence, we require from the user a vision of the ranges of possible values of formula computations. This enables us to perform interval analysis on spreadsheets to detect the existence of faults in formulas.

Chapter: 4
Related Work

Realizing the quality problems spreadsheets have, different approaches to improve spreadsheet quality have been proposed. This chapter surveys approaches to improve spreadsheet quality. The approaches can be broadly categorized as preventive and detective. Though the research most relevant to this work is the detective approach, we would like to review both approaches to give insight into the state of the art. Section 4.1 reviews preventive approaches. A discussion and evaluation of detective approaches is given in section 4.2. Finally, section 4.3 concludes the chapter by presenting a summary of the spreadsheet quality improvement approaches.

4.1 Preventive Approaches

The focus of preventive approaches is to provide a design methodology. A number of approaches have attempted to introduce design styles one way or the other. This section reviews the main preventive approaches that have been discussed in the literature. It covers approaches that apply concepts of software engineering as well as approaches that provide mechanisms of redesigning the way formulas are defined. Section 4.1.1 discusses approaches which adapt conventional software design methodologies for spreadsheets. At a lower granularity, section 4.1.2 discusses approaches targeted to formula design. Section 4.1.3 presents an evaluation of the preventive approaches.

4.1.1 Spreadsheet Design Approaches

Different authors have indicated the lack of a development methodology for spreadsheets and considered this as one of the causes for high error rates [63, 82]. Panko indicated that the problem in spreadsheet development resembles the problem in conventional software before the introduction of structured programming. While an ad hoc style of development is widely used to speed up the development process, the quality of the resulting spreadsheet suffers. To tackle the problem from a design perspective, different approaches have been proposed to improve the way spreadsheets are designed.

Ronen et al. [97] indicated the necessity of a formal analysis and design approach and to this end demonstrated a structured design technique for spreadsheet programs. They proposed a development life cycle for spreadsheets where different components of a spreadsheet can be identified. Block-structure diagrams are used to represent input vectors, output vectors, decision vectors, parameter vectors, and formulas separately. The relationships among the different components are represented using dataflow diagrams. In a similar manner, Mather [63] suggested a framework for developing spreadsheet programs. This framework provides the identification of input, output, and intermediate variables. This approach also uses different symbols for the representation of input, output, and intermediate variables. It also provides some guidelines how to structure the program on the two dimensional grid structure by separating constants area, calculation area, and manager area (final outputs area).

Knight et al. [53] proposed a design methodology for spreadsheets based on Jackson structure diagrams. The approach provides a modularization principle to decompose a spreadsheet program into modules. The main constructs of Jackson structure diagram sequence, repetition, and selection are applied to construct a structure diagram for a spreadsheet module. Once a spreadsheet design is expressed based on Jackson structure diagrams, it can be represented on the tabular grid in an indented form which provides a structured layout. They have also indicated the potential of the approach to transform existing spreadsheets into Jackson structure diagrams (reverse engineering approach).

Isakowitz et al. [49] argue that a spreadsheet program is manipulated as one entity while it contains different components that require different representations. A spreadsheet program, according to them, can be viewed from two perspectives: logical and physical. Using database terms, the logical view consists of schema and data components. The schema is a formal description of the program's logic (formulas and their relationships to each other). The data contains a structured set of values on which the schema property operates. The physical view consists of the physical layout of the program on the grid structure. This comprises editorial and binding. The editorial contains anything left after the extraction of schema and data. These are mainly titles and comments. The binding is a mapping that binds schema, data, and editorial to the spreadsheet grid. A factoring algorithm is used to extract the schema with the help of the user from a spreadsheet program. The reverse process of constructing a spreadsheet program from a schema is accomplished using a synthesis algorithm. To perform the extraction of schema and data, a certain structural assumption is taken in this approach. In addition, the user is required to identify the relations (block of related cells) out of which the schema will be constructed.

A similar approach to Wilde [120] is presented by Tukiainen [113] which is called ASSET. The principal idea of this approach is to change the design style of spreadsheets so that the representations of cells and their relations are more understandable. This approach focuses on the problems of invisibility of formulas and the physical distance between a formula and cells referenced by the formula. In ASSET, a user creates a spreadsheet program by choosing the type and size of data structures appropriate for the computational task. The data structures are chamber, sequence, and table which are introduced to simplify the way data is organized. A chamber consists of a single cell. A sequence is a one-dimensional collection of cells. A table is a two-dimensional collection of cells. For example, a user can choose a sequence structure with 10 elements to put the values of 10 items. After filling the cells with data, a goal is selected based on the desired computation. Goals and plans, which are used in conventional programming, are also found to be used by spreadsheet programmers.

Computation is performed by applying goals and plans. Goals define what must be accomplished to solve the problem, and plans define how the goals can be achieved. Goals and plans are implemented as predefined functions and are available during programming a spreadsheet. ASSET enables a user to group cells in a logical collection. The logical collection consists of those cells contained in a sequence or table structure. This avoids unintended reference to other cells. In addition, whenever a new insertion of data is made within a given logical group of cells, the computation is automatically updated irrespective of the position of insertion. This solves the problem with the current spreadsheet systems where a new insertion at the border of a range is not automatically included in the computation. This way of computation forces the user to plan ahead the number of rows and columns needed to layout data.

An approach from an object-oriented software perspective has been proposed by Paine [80] which requires the user to write textual specifications of objects which are needed to build a spreadsheet program. The object specifications are then mapped to a tabular grid using an intermediate compiler called MM (model Master). In essence the user is expected to write an object-oriented version of his/her spreadsheet program and then with the help of MM transform it into the desired spreadsheet program. As demonstrated, errors due to cell misreference (e.g., referencing a wrong cell in a formula) and type mismatch could be reduced.

A different approach is to provide users a training on systems analysis and design methods. Kreie et al. [56] claim that lack of training on analysis and design could be one of the reasons for high error rates. In addition, they have indicated that if users receive a training on such topics, they will likely apply the concept when designing spreadsheets. In their study, they found that the overall design of spreadsheet programs developed by end users who received analysis and design training improved. A similar study [51] indicated the importance of structured design approach to spreadsheet development. Chadwick et al. [18] also indicated that high error rates in spreadsheets are due to failure to consider spreadsheets as software. They argue that spreadsheets are software and need application of concepts of software engineering to the design and development of spreadsheets. They proposed a life cycle approach for spreadsheet development called R.A.D.A.R which encompasses requirements, analysis, design, acceptance, and review. The subjects of the study were university students who developed spreadsheets using the concepts of modularization and applied the life cycle approach to spreadsheets. Their result shows that spreadsheet quality can be improved by applying software engineering concepts.

4.1.2 Formula Design Approaches

The other direction of tackling the problem is to make design at a formula level. Wilde [120] proposed redesigning the way formulas are defined. In his approach, computations are built up by specifying operators on a nearby range of cells. The cell containing the computed value and the cells up on which computation is performed are contained within a specified cell range. There are no formulas in the sense of the usual spreadsheet. Computations are more visible since they are defined in a nearby area.

In an attempt to represent formulas visually, Cox and Smedley [25] proposed a way of redefining the way formulas are defined. In their approach, computation is performed by representing formulas by a graphical dataflow diagram. Cell values involved in the computation are associated to dataflow graph nodes. Arrows from each node of the dataflow graph to the corresponding cells of the spreadsheet are drawn to show which cells are used in the computation. In addition, operators and functions are also attached in the dataflow graph at appropriate positions so that the semantics of the computation is understood from the dataflow diagram. The sink node of the dataflow graph displays the result of the computation. This dataflow graph representation of a formula clearly shows the cells involved in the computation, operators and functions used in the formula. It could be also helpful in debugging a formula.

4.1.3 Evaluation of Preventive Approaches

Preventive approaches tackle the problem from two perspectives. The first perspective is adaptation of conventional software development process to spreadsheet development. This is accomplished by using a design technique [18, 49, 53, 97]. Design approaches help in the process of creating spreadsheets and the resulting spreadsheet will be easily understandable and maintainable. In addition, during the process of the design, flaws can be identified and corrected before propagated to the code. This reduces the cost of detecting and correcting faults. However, as spreadsheet systems are used by a large number of non-professional programmers, formalized development methodologies for spreadsheets are unlikely to be used by a significant proportion of users. One of the reasons for the wide acceptance of spreadsheet systems is that they can be used without any formal training on conventional software development process. Moreover, users develop spreadsheets very quickly as compared to the development of conventional software. Therefore, to gain popularity, design approaches should not force the user to change the way of spreadsheet development as users may not accept design procedures which will reduce their productivity. The adaptation of conventional software design methods to spreadsheets should mainly consider the developers of spreadsheets.

The other perspective is changing the way formulas are defined [25, 80, 113, 120]. This approach seems promising provided that the way formula design is accomplished doesn't require much concept of procedural programming. For example, the way a formula is specified in [25] is more visible and helpful for further examination of cell dependencies. On the other hand, Paine [80] requires writing an object-oriented program to build a spreadsheet program. Though it can help to alleviate some of the problems, an approach that requires writing an object-oriented program will miss most spreadsheet users. Similarly, Wilde [120] restricts the type of formulas which can be created and as a result its usefulness is very limited. ASSET [113] provides a structured way of laying out data and the relationships among group of cells are more visible. However, all computations cannot be performed using predefined goals and plans. It should be flexible enough to allow users to define their own formulas in some way. If creating ones own formula requires a sort of conventional programming, then it is unlikely that a user of a spreadsheet will afford that.

All the preventive approaches have the objective of reducing errors by providing preventive mechanisms. Preventing errors is cost effective but cannot stop all errors in a program. Hence a detective mechanism is needed to detect those errors which cannot be prevented. Preventive and detective approaches are complementary in nature.

4.2 Detective Approaches

Faults which are not prevented during the earlier stages of development can only be identified through detective mechanisms. In spreadsheet development where there is no well defined and user-accepted development process, detective approaches are highly desirable. Detective approaches have the objective of showing a symptom of fault in a spreadsheet program. This can be achieved through visualization and testing tools. Section 4.2.1 discusses approaches proposed to provide a graphical representation of a spreadsheet program. Section 4.2.2 reviews testing techniques that are targeted to spreadsheet programs. An evaluation of detective approaches is presented in section 4.2.3.

4.2.1 Visualization

One of the difficulties in understanding spreadsheet programs is the invisibility of the cell dependencies. Visualization approaches provide a visible graphical representation of the computational structure of a spreadsheet. The graphical representation can be in terms of arrows, colors, shades, boxes etc. which show groups of related cells and their interactions. Besides showing the possible faults, visualization tools have also the potential of making program understanding easier.

The spreadsheet detective [8] adds colored annotations to spreadsheets so that formula cells can easily be identified from input cells. For example, overwriting a formula cell with a fixed value can be easily identified. In addition, it provides shading patterns for formula cells which show the direction (vertical or horizontal) from which a formula is copied. It also provides a mechanism of comparing versions of a spreadsheet by identifying the changed formulas and rows(columns) inserted (deleted). The changed formula cells and inserted (deleted) rows (columns) are marked with different symbols.

Most visualization approaches try to extract cell dependencies and display them in a graphical form. In [106], a data dependency graph is presented in a three-dimensional space. While the spreadsheet layout itself is two-dimensional, the third-dimension is used to depict inter-cell dependencies. A desired formula cell is lifted up in the z-axis by some units. As this cell is lifted up to higher levels, the cells upon which the selected cell depends or the cells which depend on the selected cell or both will be lifted up one level less the level of the selected cell. This way, cell dependencies can be examined without superimposing arrows and colors on the spreadsheet itself.

Fluid visualization [48] applied the idea of animation to spreadsheet visualization. Different techniques are used to visualize the dataflow structure either partly (i.e., cell by cell) or globally (i.e., entire dataflow structure at once). The cell by cell visualization displays the precedent and dependent cells of a given cell with links and shades as the user moves the mouse pointer over the desired cell. On the other hand, when the user requires a general overview of the entire dataflow structure at once, the dataflow graph is displayed with different shades and colors showing the cell dependency relationships. This is also provided in an animated form which shows the dataflow structure in a progressive way (i.e., from input cells through intermediate cells to output cells). Moreover, fluid visualization provides an interactive feature to navigate and edit a dataflow graph.

Besides the dataflow graphs, spreadsheets will be better understood if a mechanism of decomposition into different functional parts is also provided. In [21], different techniques are used to incorporate functional parts (a kind of modules) in the visualization of a spreadsheet. The first identification is made between input, intermediate, and output cells based on the availability of precedent and dependent cells. Anomalous cells which have neither precedents nor dependents are easily identified . Similar to cell connections, block connections are also drawn to identify the interactions among different groups of cells. The variety of techniques demonstrated provide different levels of focus from cell by cell links to links among blocks of cells.

A more general approach to spreadsheet visualization is proposed by Sajaniemi [102]. He described a theoretical model of spreadsheets upon which visualization mechanisms are introduced. This visualization approach also treats areas and introduces different criteria to identify areas. An area contains formulas or input values which form a logical entity. An area which contains a homogeneous collection of cells is treated as a single entity and as a result there is a single arrow emanating from and/or heading into such an area. A homogeneous area is an area which contains a group of cells which are similar. Two cells are similar if they are type and formula equivalent. In addition, different types of areas such as top-originating, bottom-originating, edge-originating, and corner-originating are used to categorize groups of cells into single entities based on their computational functionalities. Areas are colored yellow with a thick border line and connected by arrows when there are interactions among the areas. Like the other visualization approaches, the dataflow structure of a spreadsheet is displayed in a graphical form indicating cell relationships. The arrows indicate the interactions between different entities. Moreover, this approach also establishes relations between formulas (i.e., relations between source and copies of a formula).

A visualization approach which also considers physical and logical areas based on the the spatial layout and the similarity of data manipulation is given in [5]. A physical area contains cells which are usually used as input for a certain aggregation function. On the other hand, a logical area contains a group of cells showing similar functionalities.

Davis [26] proposed two auditing tools, one based on data dependency diagrams [97] and another representing data dependencies using arrows superimposed on the spreadsheet display. The on-line data dependency diagram provides a graphical layout of inputs, outputs, decision variables, parameters, and formulas with different symbols showing the data dependency among cells using arrows. The diagram is displayed in a separate window from the spreadsheet display. The correspondence between a cell and its diagram is indicated by using the same color whenever needed. The arrow tool shows data dependencies by drawing arrows and colors between precedent and dependent cells. The tool also provides a means of navigating the arrows. Based on empirical evaluation of Davis's arrow and on-line data dependency diagram tools, the arrow tool is found to be more appropriate for investigating cell dependencies and helpful for debugging. Microsoft Excel's auditing tool is similar to Davis's arrow tool which is intended to depict relationships between cells and help in debugging spreadsheets. This tool draws arrows from precedent cells to dependent cells indicating the direction of dataflow. If the precedent cells are groups of cells which form an area, then an enclosing rectangular border is drawn around the area. A drawback of this style of arrow representation is that it is difficult to identify cell dependencies when there is an overlap of arrows. In addition, when there are many dependencies, a lot of arrows are drawn which makes tracing relationships difficult and the display gets cluttered.

Even though visualization approaches have the objective of presenting the hidden dataflow structure in some graphical form, they vary in features and procedures for investigating (i.e., browsing data dependencies) precedent and dependent cells. They also differ in the way the graphical form is displayed. Some provide a separate graphical display (e.g., [26, 106]) while others superimpose it on the spreadsheet display (e.g., [21, 48, 102]). Separate representation has the advantage of viewing both displays without much information at the same display; but determining the correspondence between separate displays adds some extra work for the user. Superimposing simplifies the task of determining the correspondence between the two displays, but both representations may not be seen clearly at the same time. However, the appropriateness of the display type (i.e., separate representation or superimposing) cannot be easily determined by analytical methods.

Generally, the differences between visualization approaches can be examined from different perspectives such as the ability to incorporate the concept of areas, the ability to incorporate formula relationships, and how easily navigation and zooming (i.e., different levels of view) can be done.

4.2.2 Testing

These approaches try to indicate the existence of faults in a spreadsheet by applying similar techniques of conventional software testing. However, the literature is meager in this direction. To our knowledge there have been only few attempts made to tackle the problem from this perspective. The only testing approach known to us in the literature is the one proposed by Rothermel et al. [99, 100]. This approach introduces the idea of the all-uses dataflow adequacy criterion to spreadsheets. Adequacy criterion helps to select test cases and provides information on the adequacy of test cases. Cells are treated as variables of a procedural program. Using dataflow testing terms, input cells are taken as variable definitions and formula cells as both definitions and uses. As the user enters different values to input cells (each input being taken as a different test case), the testing system records the def-use associations executed during validation. The user is supposed to validate each formula cell under test as correct or incorrect for a given test case. Based on this, the degree of testedness of a formula cell is computed by comparing the ratio of executed def-use paths to the total def-use paths of that formula cell. Whenever the need arises, the user is provided with information about the degree of testedness of a formula cell. The assumption is that if the user sees a lower degree of testedness of a formula cell, (s)he will test with additional test cases. In addition, whenever the user changes a formula, the testing system provides feedback about how this affects the testedness of those dependent cells.

The drawback of this approach is the assumption that users will provide test cases to satisfy the all-uses dataflow adequacy criterion without knowledge of the underlying dataflow structure of the code (i.e., def-use graph). Understanding the dataflow structure is more difficult in spreadsheets where code is represented in two-dimension and input/program/output are not represented separately. As the testing criterion used remains implicit (users are not aware of it), the tests they generate are random which may not provide adequate coverage. The weakness of random testing is discussed in section 3.4 of chapter 3. Since users are not aware of the complications of achieving adequate code coverage, they may not be convinced by a test coverage information below full coverage. For example, for a given formula, if the degree of testedness is say 30% def-use coverage, then it will be difficult for the user to provide additional test cases to achieve 70% or 80% def-use coverage without knowing which def-use paths are executed. Had (s)he been a professional programmer, (s)he would design new test cases for the def-use paths which are not yet exercised by any test case.

This approach was originally designed to be used for testing each formula cell separately. As an improvement, a scale up of the approach [13] was proposed to be able to test a group of cells at once. The new version has two approaches: straightforward and region representative. In the straightforward approach, the user can validate a group of cells in one operation but testedness information is maintained separately for each cell. In the region representative approach, a single formula cell is selected and tested and the testedness information is propagated automatically to those formula cells in the region. This simplifies the effort needed to test each formula cell and as a result the number of test cases needed is greatly reduced. In addition, the user is not required to specify the expected value for each formula which needs to be tested. However, this approach can also lead to wrong conclusions of testedness of formula cells when a formula is copied to a group of cells. One very common task in spreadsheet programming is copying a formula from one cell to a group of cells. Depending on the cells to be referenced by the formulas in the new locations, relative and absolute references are used. If an error occurs in setting relative and absolute references in the source formula, then the copied formulas refer to other cells which are not intended for the computations. Let us consider the example in figure 4.1 to illustrate this situation.

Suppose a user specified a formula in cell D2 and copied it across the cells D3 to D5. Since the column should not change during copying, an absolute reference is used for the column. But as the formula is copied down to the cells D3 to D5, the row should vary. However, the user made an error and used an absolute reference for the row too. The formula is correct for the first formula cell D2 but the formulas in cells D3 to D5 refer to the wrong tax rate cell B2. As a result the values of the cells in D3 to D5 are incorrect. According to the region representative approach, the user is likely

	A	B	C	D
1	**Year**	**Tax Rate**	**Sales**	**Tax Amount**
2	1991	0.15	3000	=$B$2*$C2
3	1992	0.2	5000	=$B$2*$C3
4	1993	0.2	4800	=$B$2*$C4
5	1994	0.3	6000	=$B$2*$C5
6				
7				

Figure 4.1: Formula copy

to test the source formula D2 and propagate the testedness information to the copies of the formula. Even though the source formula appears to be correct, its copies are not correct. Due to the absolute reference which refers to a fixed cell during copying, the copies have unintended cell references. In such cases, propagation of testedness information provides false information. Hence, the generalization approach can fail due to errors in setting cell reference types.

The other approach proposed in spreadsheet testing is code inspection. Code inspection involves the examination of program code to detect faults. Panko [82] has conducted an experimental study on the effectiveness of code inspection to spreadsheet testing. In his study, it was found out that individual code inspection detected 63% of the faults while group inspection detected (a team of three inspectors) 83% of the seeded faults. The result is in line with the effectiveness of code inspection in conventional software testing. One advantage code inspection gives over dynamic testing is that code inspection detects the faults themselves directly whereas dynamic testing provides only symptom of faults. However, code inspection cannot be automated and hence cannot be used by individuals to test their own programs. Therefore, code inspection is a matter of policy setting, preparation of guidelines and team organization instead of being used as a tool.

4.2.3 Evaluation of Detective Approaches

Detective approaches have the objective of showing symptoms of faults in a spreadsheet program. The way a symptom of fault is generated depends actually on the type of detective approach used. Generally, visualization approaches are concerned with the layout of data and the corresponding dataflow structure while testing approaches are concerned with formulas defined for computations.

Visualization tools are used to examine the validity of spreadsheet programs by tracing the interactions among cells to uncover unintended or missing connections. They indicate irregularities or mismatches between the physical structure and the invisible dataflow structure. However, visualization approaches do not focus on faults within the formulas; they rather try to highlight anomalies by investigating relationships among cells that show existence of potential errors. On the other hand, testing approaches are concerned with the detection of faults within formulas. Code inspection approaches enable the user to detect the faults in formulas directly. Nevertheless, this is mainly a code reading activity which does not provide the user an additional support to detect faults. Dynamic testing, on the other hand, provides a symptom of fault which aids the user to examine those cells for which a symptom is reported. For the identification of the possible location of faults, the data dependencies are traced using visualization (or debugging tools). We can see that visualization and testing approaches are complementary since they focus on different types of faults. The existing testing approach requires the user to provide tests to satisfy a test adequacy criteria which was defined for conventional software testing. Adaptation of such techniques could be helpful, but should take into consideration the inherent characteristics of spreadsheets and their developers.

4.3 Summary

The reasons for spreadsheet errors could be attributed to the lack of a well defined design and testing support. A design methodology helps to reduce the faults which will be introduced into the program and as such it is a preventive mechanism. However, this by itself does not ensure the quality sought for. This process has to be accompanied by a testing phase which is targeted to the detection of latent faults.

There are two broad categories of approaches aimed at improving spreadsheet quality: preventive and detective approaches. Preventive approaches try to improve quality by providing better design mechanisms. The objective of preventive approaches is to prevent errors that can be prevented before they are introduced into a spreadsheet program. Some approaches propose a development life-cycle for spreadsheets; others provide a way of defining formulas. Some of the design approaches force the user to change the way of spreadsheet development and others assume some concepts of conventional software development to be used directly by a spreadsheet developer. Though preventive approaches are very helpful in that faults can be identified at earlier stages, they should neither require concepts of conventional software development nor force the user to change the style of spreadsheet development.

Detective approaches which consist of visualization and testing approaches try to provide techniques of detecting the existence of faults. Visualization approaches indicate the existence of potential faults in a spreadsheet program by providing a graphical representation based on the data flow structure. However, visualization approaches do not provide mechanisms of detecting faults within a formula. Testing approaches attempt to indicate the existence of faults within formulas. The drawback of the existing testing approach is the assumption that users will provide test cases to satisfy the all-uses dataflow adequacy criteria without knowledge of the dataflow structure of the code (i.e., def-use graph) and the testing criterion used. Since users are not aware of the testing criteria employed, the tests they generate are random which may not provide adequate coverage. In addition, users are working on actual data that they need for their applications and need to know the correctness based on the actual data instead of arbitrary data chosen for testing purpose.

Rather than addressing the problem of spreadsheet quality from a preventive aspect (i.e., preventing errors), we focus on tackling the problem from a detective aspect (i.e., detecting existence of faults) without requiring the user to learn software engineering concepts. In chapter 6, we present a testing methodology for spreadsheets based on interval analysis and symbolic testing. The use of interval-based testing does not force the user to change the way of spreadsheet development.

Chapter: 5
Basic Interval Arithmetic

This chapter reviews the basic definitions and properties of interval arithmetic which are helpful with regard to spreadsheet testing. Interval-based testing methodology thus works based on these definitions and properties and by adapting to spreadsheet interval computation whenever necessary. Section 5.1 introduces the basic idea of intervals. Section 5.2 presents the most common definitions of interval arithmetic for single and multi-intervals. A treatment of relational operators for single and multi-intervals is presented in section 5.3. Finally, a short summary is given in section 5.4.

5.1 Intervals

An interval represents a range of possible values bounded by the interval's lower and upper bounds. Interval arithmetic was introduced by Moore [68] and is used for a variety of applications such as solving differential equations, linear systems, and global optimization. It is used to evaluate arithmetic expressions over sets of numbers contained in intervals. The result of interval computation is an interval that is guaranteed to contain the set of all possible resulting values bounded by the global minimum and global maximum. Interval arithmetic is also used for error analysis as numerical computations involve different errors. The errors could be due to errors in the data, rounding errors, or approximation errors. Interval analysis is used to deal with uncertain data, rounding errors, and approximations and as a result to provide bounds on the effect of such errors on a computed quantity [36].

Classical interval arithmetic deals only with closed intervals. However, in practical situations open intervals and intervals with infinite end points are common. To incorporate this situation, extended interval arithmetic is used. However, we will use mostly classical interval analysis for spreadsheet evaluations even though extended intervals can also be introduced.

5.2 Interval Arithmetic

The classical interval arithmetic is usually defined for single intervals. However, in the case of division of intervals containing zero value, the resulting interval could be union of intervals which involves infinity end points. The resulting interval can also be used for further interval computation. Therefore, it is also necessary to define arithmetic operation on multiple intervals.

5.2.1 Single Intervals

Let X = [a, b] and Y = [c, d] be intervals and *op* represent arithmetic operators +, -, *, /. Then $X\,op\,Y$ = [min x *op* y, max x *op* y] where x $\in$ X and y $\in$ Y. The min and max are global minimum and maximum values. Based on this definition, the following basic interval arithmetic operations are defined [36, 45].

- $X + Y = [a + c, b + d]$
- $X - Y = [a - d, b - c]$
- X * Y = [min$\{ac, ad, bc, bd\}$, max$\{ac, ad, bc, bd\}$]
- $\frac{1}{X} = [\frac{1}{b}, \frac{1}{a}]$ provided that $0 \notin X$
- $\frac{X}{Y} = X * \frac{1}{Y}$, $0 \notin Y$

Division of intervals can be generalized as follows:

$$\frac{X}{Y} = \begin{cases} [a,b] * [\frac{1}{d}, \frac{1}{c}] & \text{if } 0 \notin [c,d] \\ \{\} & \text{if } c = d = 0 \\ \frac{[a,b]}{(0,d]} & \text{if } c = 0 \\ \frac{[a,b]}{[c,0)} & \text{if } d = 0 \\ \frac{[a,b]}{[c,0)} \cup \frac{[a,b]}{(0,d]} & \text{otherwise} \end{cases} \tag{5.1}$$

As real numbers are degenerate intervals, it is also possible to perform arithmetic operations on combinations of both real and interval numbers. The following operations define mixed arithmetic. Let $k > 0$ (the case for $k < 0$ can be defined similarly) be a real number.

1. $k + X = [k + a, k + b]$
2. $k - X = [k - b, k - a]$
3. $k * X = [k * a, k * b]$
4. $\frac{k}{X} = k * \frac{1}{X}$
5. $\frac{X}{k} = [\frac{a}{k}, \frac{b}{k}]$

Other interval functions which are commonly used are absolute value and integral exponent of an interval. These definitions are given below [36]. Let $X = [a, b]$ be an interval.

- Absolute value

$$|X| = \begin{cases} \text{[a, b]} & \text{if } a \geq 0 \\ \text{[-b, -a]} & \text{if } b \leq 0 \\ [min\{|a|, |b|\}, max\{|a|, |b|\}] & \text{if } a < 0 < b \end{cases} \tag{5.2}$$

- Integral exponent

$$X^n = \begin{cases} [1, 1] & \text{if n} = 0 \\ [a^n, b^n] & \text{if } a \geq 0 \text{ or if } a \leq 0 \leq b \text{ and n is odd} \\ [b^n, a^n] & \text{if } b \leq 0 \\ [0, max(a^n, b^n)] & \text{if } a \leq 0 \leq b \text{ and n is even} \end{cases} \tag{5.3}$$

So far we have considered only closed intervals. A similar definition can be used for open intervals (see [45]). Infinite intervals need a special consideration. The rules for addition and subtraction of infinite and semi-infinite intervals are given below.

1. $[a, b] + (-\infty, d] = (-\infty, b + d]$
2. $[a, b] + [c, \infty) = [a + c, \infty)$
3. $[a, b] \pm (-\infty, \infty) = (-\infty, \infty)$
4. $[a, b] - (-\infty, d] = [a - d, \infty)$
5. $[a, b] - [c, \infty) = (-\infty, b - c]$

5.2.2 Multi-Intervals

As seen in the case of interval division containing the value zero, the resulting interval could be a union of intervals. Such intervals are called multi-intervals (or discontinuous intervals). If multi-intervals are involved in further computations, then arithmetic on multi-intervals is necessary. For example, if X, Y, and Z are intervals, then $(X \cup Y) \pm Z = (X \pm Z) \cup (Y \pm Z)$. The basic arithmetic for multi-intervals is given below.

Let $X = X_1 \cup X_2 \cup ... \cup X_n = \{X_1, X_2, ..., X_n\}$ and $Y = Y_1 \cup Y_2 \cup ... \cup Y_m = \{Y_1, Y_2, ..., Y_m\}$ be sets of intervals where the component intervals are disjoint.

1. Addition

   $X + Y = \bigcup_{i=1}^{n} X_i + \bigcup_{j=1}^{m} Y_j = \bigcup_{i,j=1}^{n,m} (X_i + Y_j)$

2. Subtraction
$X - Y = \bigcup_{i=1}^{n} X_i - \bigcup_{j=1}^{m} Y_j = \bigcup_{i,j=1}^{n,m} (X_i - Y_j)$
3. Multiplication
$X * Y = \bigcup_{i=1}^{n} X_i * \bigcup_{j=1}^{m} Y_j = \bigcup_{i,j=1}^{n,m} (X_i * Y_j)$
4. Division $\frac{X}{Y} = \frac{\bigcup X_i}{\bigcup Y_j} = \bigcup_{i,j=1}^{n,m} (\frac{X_i}{Y_j})$

   Note:

   (a) The set of resulting intervals may be reducible if there are overlapping intervals.

   (b) The number of constituent intervals in the resulting discontinuous interval lies between 1 and n*m.

5.2.3 Interval-valued Functions

Consider a real-valued function f of real variables $x_1, x_2, ..., x_n$ and an interval function F of interval variables $X_1, X_2, ..., X_n$. The interval function F is said to be an interval extension of f if $F(x_1, x_2, ..., x_n) = f(x_1, x_2, ..., x_n)$ for all x_i.

Generally, interval functions are interval extensions of ordinary functions. Since the resulting value of the function F contains the true range of the function f over the given interval, $f(x_1, x_2, ..., x_n) \in F(X_1, X_2, ..., X_n)$ always holds. True range of a function represents the range of values of the function over the given interval. For example, let $f(x) = (x - 1) * (x + 1)$ over [-2, 2]. Then f([-2, 2]) = [-3, 1]*[-1, 3] = [-9, 3] while the true range of the function is [-1, 3]. The reason for such a wide range is the dependency problem. Here, the variable x appeared twice in the expression and gave rise to a wide range. Hence, to find the true range of a function, other methods should be used. In general, finding the true range of an interval function is not straight forward. The true range of a function can be easily obtained if the function is monotonic. For a monotonically increasing or decreasing function f over an interval [a, b], the range is either [f(a), f(b)] or [f(b), f(a)]. In the general case, we need to use optimization methods to find the global minimum and global maximum of an interval function. As we are concerned only with the basic interval arithmetic, we do not deal much with interval functions (e.g., trigonometric, exponential, logarithmic etc.) and hence no discussion of optimization methods is provided.

5.3 Interval Relational Operators

Since intervals are not totally ordered (they have only partial order) [36, 52, 68], they may not be comparable. The set of intervals does not have Boolean comparison operators. Generally, the comparison operators of the set of intervals are three-valued operators (i.e., T, F, TF). There are different approaches used to transform into Boolean logic operators based on the context of use of the comparison operators. For example, Hyvönen [44] mentioned "constraint $X > Y$ is satisfiable if there is some x in X such that there is some y in Y such that $x > y$." Others use demotion which performs simple conversion. For example, in [116], optimistic demotion($TF \longrightarrow T$) or pessimistic demotion ($TF \longrightarrow F$) is indicated as a possibility. A better approach is used in FORTRAN 95 [66] by introducing qualifiers "Certainly", "Possibly" and "Set theoretic".

Let X = [a, b] and Y = [c, d] be intervals and $op \in R = \{<, \leq, =, \geq, >, \neq\}$. Table 5.1 taken from [66] shows the operational definitions of $X op\, Y$.

	Set	**Certainly**	**Possibly**
$<$	$a < c \wedge b < d$	$b < c$	$a < d$
$\leq$	$a \leq c \wedge b \leq d$	$b \leq c$	$a \leq d$
$=$	$a = c \wedge b = d$	$d \leq a \wedge b \leq c$	$a \leq d \wedge c \leq b$
$\geq$	$a \geq c \wedge b \geq d$	$a \geq d$	$b \geq c$
$>$	$a > c \wedge b > d$	$a > d$	$b > c$
$\neq$	$a \neq c \vee b \neq d$	$a > d \vee c > b$	$d > a \vee b > c$

Table 5.1: Operational definition for interval comparison

The operational definition given in table 5.1 is based on the following mathematical definition. Let X and Y be non-empty intervals and $op \in R \setminus \{\neq\}$ where R is as defined above.

- Set theoretic

  X op $Y \equiv \forall x \in X, \exists y \in Y : x$ op y and $\forall y \in Y, \exists x \in X : x$ op y.

- Certainly

  X op $Y \equiv \forall x \in X, \forall y \in Y : x$ op y.

- Possibly

  X op $Y \equiv \exists x \in X, \exists y \in Y : x$ op y.

Note:

- Certainly equal holds iff the two intervals are degenerate intervals (i.e., intervals of width zero) and possibly equal holds iff the two intervals are non-disjoint.

- Possibly not equal holds iff the two intervals are not degenerate intervals.

Based on the definitions given in [36, 66], we define comparison operators for single and multi-intervals as follows.

1. Single Intervals

   - $X = Y$ iff $a = c \wedge b = d$
   - $X \neq Y$ iff $a \neq c \vee b \neq d$
   - $X \geq Y$ iff $a \geq d$
   - $X \leq Y$ iff $b \leq c$

2. Multi-intervals

   Since the comparison of multi-intervals requires normalization, we define first normalization of multi-intervals. The normalized form of a multi-interval as given in [45] satisfies the following conditions.

(a) the component intervals are single intervals

(b) any two intersecting components are merged into a single interval

(c) the components are sorted in increasing order

To define multi-interval comparison, let $X = X_1 \cup X_2 \cup ... \cup X_n = \{X_1, X_2, ..., X_n\}$ and $Y = Y_1 \cup Y_2 \cup ... \cup Y_m = \{Y_1, Y_2, ..., Y_m\}$ be normalized multi-intervals.

- $X = Y$ iff $\forall X_i \in X, \exists Y_j \in Y : X_i = Y_j \wedge \forall Y_j \in Y, \exists X_i \in X : X_i = Y_j$
- $X \neq Y$ iff $\exists X_i \in X, \forall Y_j \in Y : X_i \neq Y_j \vee \exists Y_j \in Y, \forall X_i \in X : X_i \neq Y_j$
- $X \geq Y$ iff $[minX_1, maxX_n] \geq [minY_1, maxY_n]$
- $X \leq Y$ iff $[minX_1, maxX_n] \leq [minY_1, maxY_n]$

5.4 Summary

Intervals represent the possible range of values bounded by the minimum and maximum. Computations are made on intervals to estimate the range of the possible outcomes. Interval arithmetic is used to evaluate arithmetic expressions over sets of numbers contained in intervals and yields an interval which usually contains the global minimum and global maximum among the possible values of the computation.

Since intervals are easily understandable and can be appropriate to represent uncertain numerical data, interval-based testing uses intervals to represent input domains and expected intervals for input and formula cells respectively and to perform interval arithmetic based on the formula defined for a cell. Interval arithmetic and comparison can be performed on both single and multi-intervals. Single interval arithmetic and comparison will be used for ordinary spreadsheet formula computation while multi-interval arithmetic and comparison will be used for spreadsheet formulas which involve alternatives and result of division.

Since we are introducing intervals to express expected ranges of values, we don't expect the user to be so precise and define open intervals instead of closed intervals.

For this reason we do not deal with open intervals except when they are results of computations such as division. The interval arithmetic we are interested in is mainly classical interval arithmetic. Similarly, if zero appears as one of the end points of an interval which is used as a divisor during division, we approximate it by the smallest or largest number next to zero depending on the direction of inclusion.

Chapter: 6

Interval-based Testing Approach

Spreadsheet programs, artifacts developed by non-programmers, are used for a variety of important tasks and decisions. Yet a significant proportion of them have severe quality problems. However, very little is known about the use of testing as a means to improve spreadsheet quality. This chapter discusses the interval-based testing approach which addresses the problems in testing spreadsheet software.

An introduction which provides an overview of the interval-based testing approach is presented in section 6.1. Section 6.2 discusses the basis for the interval-based testing approach. A discussion on the constituents of a spreadsheet program test process is given in section 6.3. To verify the reasonableness of expected intervals provided by the user, interval analysis is performed on spreadsheets. A discussion on how to attach input and expected intervals and how to perform interval computation on spreadsheets is given in section 6.4. Section 6.5 discusses the conditions and procedures on how to generate symptoms of faults. After testing is performed, it is important to get a general information regarding the extent of verification. Verification information for a spreadsheet program is presented in section 6.6. A real spreadsheet to demonstrate the process of spreadsheet testing is given in section 6.7. Finally, a summary of the main points of the interval-based testing approach is given in section 6.8.

6.1 Introduction

The interval-based testing approach incorporates different techniques. Being a testing methodology for spreadsheet software which is mainly used for numerical computations, it uses the idea of interval analysis. On the other hand, being a testing technique, it resembles symbolic testing which was defined for testing conventional software.

Structural testing techniques provide a mechanism of determining the validity of a unit of a program based on individual test cases. On the other hand, symbolic testing provides a mechanism of determining the validity of a unit of a program for any possible values of the input variables. Similar to symbolic testing, in the interval-based testing, the determination of the validity of formulas is based on intervals. However, unlike symbolic testing which requires expressing formulas only in terms of input variables, interval-based testing uses intermediate variables for the purpose of narrowing down the computed interval.

Interval-based testing is based on the observation that spreadsheet software are mainly used for numerical computations. This enables us to introduce the idea of interval analysis to spreadsheet software testing. This approach requires the user to specify input intervals and expected intervals to validate a spreadsheet program. Based on the intervals provided by the user, interval computation is performed for formula cells. To detect the existence of symptoms of faults within formulas, comparison is made between the values of spreadsheet computation, user expectation, and interval computation. In order to carry out interval computation and to reason about the correctness of spreadsheet formulas, there are two options which can be used during interval-based testing.

1. Strictly following the idea of symbolic testing, intervals can be specified only for input cells and all computations of bounding intervals for formula cells will be based on the intervals specified for input cells. This works fine as long as the given formula is defined by the user only in terms of input cells. However, if a formula is dependent on other formula cells, the resulting computed interval

will be wide as interval arithmetic provides the global minimum and global maximum of all possible computations. In such cases, the comparison between the expected interval specified for the given formula cell and the computed interval may not signal the existence of a symptom of fault when there is actually a fault. As a result, it is necessary to use narrower intervals for the intermediate formula cells. This can be achieved by considering the expected intervals of the intermediate formula cells instead of using the computed intervals of the intermediate formula cells.

2. The other alternative is the use of expected intervals of intermediate formula cells during the computation of the bounding interval of a formula cell which depends on one or more intermediate formula cells. This option enables us to get relatively narrower computed intervals and hence the comparison between the expected interval specified for the given formula cell and the computed interval may provide appropriate signal regarding the existence of a symptom of fault. Therefore, this option is chosen for the computation of the bounding intervals of formula cells.

While introducing the idea of intervals in the interval-based testing, there are some assumptions made.

- Users do not have exact values in mind about the results of formulas but they have some range of reasonable values which can be expressed as intervals.
- During the comparison of spreadsheet computation, user expectation, and interval computation, boundary violations are only indicators of symptoms of faults.
- Errors within the boundaries of expected intervals can go unnoticed but might not harm dramatically.
- Ripple effects might lead to noticeable boundary violations.

The interval-based testing approach presented here differs from the testing approach discussed in chapter 4 section 4.2.2 in a number of ways. The first difference is that interval-based testing is not based on code coverage (i.e., no adequacy criterion is defined). Most users are not aware of conventional software testing techniques. In addition, they are not expected to provide effective test cases to satisfy a given classical coverage criterion without knowledge of the code structure and without knowledge of the coverage criterion employed behind (as is done in WYSIWYT [99, 100]). The interval-based testing focuses on the functionality of spreadsheet formulas instead of the code structure of a spreadsheet program. Hence, other concepts have to take care of coverage of formula verification. Verification information, similar to "test coverage", can be inferred from the number of formula cells which are verified and correct with respect to the total number of formula cells (see section 6.6).

Secondly, the interval-based testing requires the expected intervals to be documented as part of the spreadsheet whereas WYSIWYT requires the expected values to be just in the minds of users and used during validation. The WYSIWYT approach requires the user to simply validate a formula as correct or incorrect based on a given test case. To do so, the user should have the expected interval in mind for a given formula. Thirdly, in interval-based testing, the expected intervals provided by the users are verified for reasonableness using interval analysis whereas the WYSIWYT approach does not provide such mechanisms.

6.2 The Rationale for Interval-based Testing

As already mentioned in previous chapters, spreadsheets are software too. But they are developed by end-users. End-user programming environments enable non-programmers to produce working applications. The easiness and usability of the environments motivate users to write their own programs. In a similar fashion, users expect a similar functionality of easiness and usability from other tools which are used to aid in the development process. As it is seen in conventional software testing, testing is an expensive process. Nevertheless, it is desirable to reduce this cost even further when it comes to end-user programming. We claim that a testing methodology for spreadsheets should take into consideration at least three points:

- `it should be user-centered`
- `it has to be spreadsheet-based`
- `it should take care of the inherent characteristics of spreadsheets`

A `user-centered approach` refers to a methodology which takes into account both the expertise of users and the conceptual models users have in mind about their programs during testing. In spreadsheet programming, the detailed procedure of computation is transferred to the language and hence is not the programmer's responsibility. A spreadsheet user is not concerned about the details of the computation. The user specifies a formula to establish dependencies so that values of other cells can be used in a computation. Users of spreadsheet systems often do not know how the result of a program is computed and hence it is unlikely that they will base their testing on the detail of computation. This is due to the fact that spreadsheet languages are declarative. Declarative languages are concerned on data relationship as opposed to control flow [4]. Users' understanding of cell dependencies is at a higher semantic level. Therefore, users will most likely check the correctness of formulas based on the intended functionality rather than their internal structure.

The coupling of input/program/output has also a similar effect. Testing by varying inputs which is the common way of testing in conventional programming is not readily applicable in testing spreadsheets. There are different reasons. Spreadsheet users develop their programs by first preparing their input data and constructing the necessary formulas to perform computations (this does not include those who develop templates). The moment formulas are specified, computations are carried out and results displayed. No need of specifying an output mechanism. As users are working on actual data that they need for their applications, it is likely that they need to know the correctness of their spreadsheets based on the actual data instead of arbitrary data chosen just for testing purposes. Input variation for the purpose of code coverage requires understanding of the detail structure of computation which is hidden from the user. Therefore, it is unlikely that testing by input variation will be effective without knowledge of the detail structure of a spreadsheet. Moreover, users prefer spreadsheet systems because they are easy to use and enable them to develop working applications in a short period of time. Thus, they may not have the time and patience to validate their programs by varying inputs (i.e., in essence generating test cases) for each formula cell. In such circumstances, they should be provided with a possibility of validating groups of cells together (i.e., `spreadsheet-based` validation may be appropriate as opposed to cell-based validation). As we will see in section 6.5, in interval-based testing, the comparison $d \in E$ is associated with spreadsheet instances whereas $E \subseteq B$ provides a more general checkup at the spreadsheet program level.

`Inherent characteristics of spreadsheets.` Spreadsheet systems are mainly used for numerical computations. During testing a spreadsheet program, developers of spreadsheet programs will have an expectation of the result of the computation of a formula. This expectation can be used to validate the formula as correct or incorrect (i.e., users have an arithmetic model). The expectation can usually be abstracted in the form of an interval which contains the possible minimum and maximum values. As a result, we require the user to specify this expected range of values of the computation in the form of intervals. Therefore, interval analysis can be performed on spreadsheets to help in detecting existence of faults in spreadsheet formulas.

Interval-based testing is similar to symbolic testing which is used for procedural programs. Actually, it can be considered as a special case of symbolic testing. Symbolic testing is used to verify a formula (i.e., symbolic output) by expressing the formula only in terms of input variables instead of using actual values. As opposed to symbolic testing where formulas are expressed only in terms of input variables, interval-based testing uses also intermediate variables (cells taken as variables) during interval computation. Symbolic testing assumes any values for the input and validity is determined for any possible value of the input variables. Unlike symbolic testing, in interval-based testing approach, the values of input variables are restricted to a range of reasonable values that can be assumed by the input variables. In addition, for the sake of narrowing down the computed interval, interval-based testing uses intermediate variables and their corresponding intervals attached by the user. For example, consider a formula E1 = D1 + C1 and D1 = A1 + B1. Since D1 itself is a formula cell, under normal condition, the bounding interval of D1 should contain the expected interval specified for it. Hence, it is legitimate to use the expected interval instead of the computed bounding interval of D1 for those cells which reference D1.

Intervals are usually used to represent an uncertain numeric data by indicating the boundaries of the possible values. Arithmetic operations performed on intervals result in intervals whose boundaries are represented by the global minimum and global maximum of the possible values of the computation. Even though global minimum and global maximum are obtained, the resulting intervals are usually wide. Hence, we need to check whether other mechanisms of narrowing the width of computed intervals can be used. We discuss the appropriateness of Scenarios and Probability distributions.

As defined in Dodge et al. [30], a scenario is a combination of values assigned to one or more variable cells in a what-if model. In the current spreadsheet systems (e.g., MS Excel 97), scenarios are generated by identifying cells which will have different(or changing) values. If numeric values of cells in a spreadsheet program are represented by intervals, a scenario can be generated based on the end points of the intervals. In such a case, a scenario can be determined by considering all combinations of minimum and maximum values of intervals [46]. Out of the possible scenarios, an interval which approximates the intended computation can be chosen. Scenarios will be helpful provided that the number of intervals used for a computation is small. The number of possible combinations of end points grows exponentially as the number of intervals increases.

A probability distribution is a mathematical function which describes the probabilities of possible events in a sample space. An interval can be seen as a probability distribution where values in the interval are chosen based on the probability distribution function used. Evaluation of functions can be performed by generating a value at a time from each interval and constructing a frequency distribution. This is typically accomplished using Monte Carlo simulation. The result is a distribution of possible outcomes and the probability of getting those results. Finally, a range of values can be chosen based on its probabilistic value. For example, DECISIONPRO [27] uses a variety of probability distribution functions for the purpose of modelling and simulation of business applications.

The enumeration of scenarios and generating a frequency distribution for the evaluation of a particular function helps in choosing an interval (with appropriate width) which serves as a better controlling mechanism of the expectation of the user. However, the number of scenarios could be large for most aggregation functions and probabilistic modeling requires some knowledge of mathematics which ordinary users may not have. Despite extra width problem, intervals are easily understandable and can be appropriate to represent uncertain numerical data [46]. Interval-based testing uses intervals to represent input domains and expected intervals for input and formula cells respectively and to perform interval arithmetic based on the formula defined for a cell.

6.3 Spreadsheet Program Test Process

After creating a spreadsheet program for a particular application, it is natural to check its correctness. We create spreadsheet programs mainly to perform numerical computations. What do we expect to be correct? For the correctness of a spreadsheet program, every input value as well as every formula should be correct. To judge the validity of the value of a formula cell, we check whether the computation is within the range of expected results. Generally, the main task in testing a program is to be able to detect the existence of faults in the program. In the context of conventional software, to achieve this we need systematically designed test cases (using an appropriate test strategy) that reveal faults in the program. By running the program with the test cases and comparing the result with the expected outcome described in the specification or generated by a test oracle, the existence of a fault can be detected. Generally, an automated test oracle requires at least some kind of specification [50, 89, 90], normally not existing with spreadsheets. Hence, mechanisms need to be devised to approach the power of a test oracle while putting minimal strains on the developers diligence and insight into complex dependencies. Thus, we must recognize that "testers" of spreadsheet programs are end-users who are not aware of testing theory and hence they are not expected to do testing in the traditional sense. Rather, users of spreadsheet systems are highly dependent on the system's assistance.

Spreadsheet testing is a comparison of the users computational model (goal) with the actual spreadsheet computation. Users usually have a computational model derived from the domain knowledge of the application they try to solve. In order to achieve the desired goal of computation, users use programming tools (in this case spreadsheet systems) as a means by which they describe the process of computation. Finally, the goal of computation which is generated by domain knowledge will be described in terms of language constructs (plans) which aid achieving the specified goal. Errors occur when the language constructs chosen do not match the desired goal or model of computation. Goal and plan as they exist in conventional programming are also found to be used in spreadsheet systems. Sajaniemi et al. [104] have conducted an experimental study and found that spreadsheet users have a set of basic programming goals and plans describing spreadsheet programming knowledge. A detailed discussion of goals and plans in spreadsheet systems can also be found in [114, 115].

Therefore, the testing process should be tackled from a different angle for spreadsheet programs. In spreadsheet programs, neither a specification is available nor spreadsheet programmers have the patience and expertise to run a lengthy suite of test cases. Therefore, interval-based testing for spreadsheets attempts to approximate the power of testing. It is based on the observation that spreadsheets are mainly used for numerical computations.

Interval-based testing focuses on numeric cells. To assess the correspondence of formulas with the spreadsheet writers intent, we compare intervals of values expected by the user with the boundaries yielded by interval computations. For the purpose of testing, in addition to the usual discrete values of numeric input cells, the spreadsheet writer attaches intervals to desired numeric input cells. This is much simpler than generating test cases (which is a very complex process especially for end users). For numeric input cells which may assume different values, the spreadsheet writer speci-

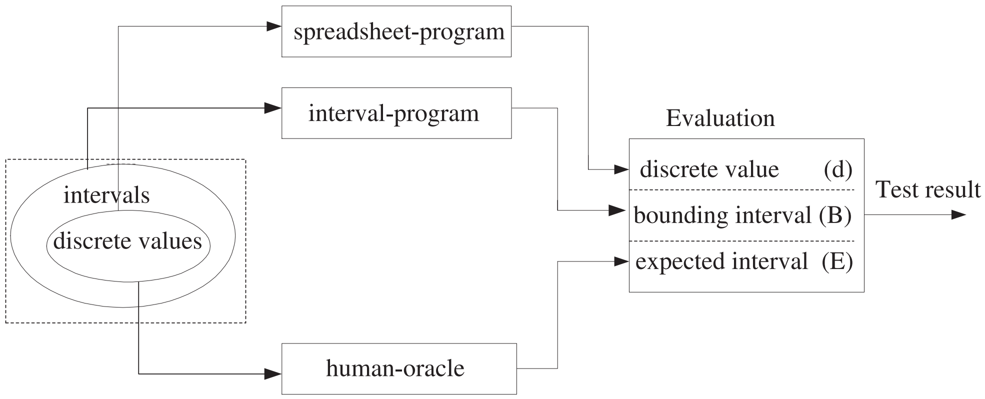


Figure 6.1: Spreadsheet program test process

fies a reasonable range of values in the form of intervals which serve as input domains. The specification of input domains is merely based on the possible values that can be used in the spreadsheet program. Some input values may be fixed numbers. Thus, they do not require interval representation. Such values can be treated as an interval of width zero.

Figure 6.1 depicts the test process for a spreadsheet program. We can see that there are three sources of information which are used to determine the existence of fault(s) in a formula cell. The information obtained from the human oracle serves as the goal whereas spreadsheet-program and interval-program provide the information based on the actual plan used to achieve the desired goal.

Based on the goal of computation, users will have a range of values in mind for the result of the computation. Therefore, the user, assuming the role of a human oracle, specifies the expected magnitude of the computation of the formula in the form of an interval for permissible/expected values. As spreadsheet developers are not expected to have formally documented specifications for their programs, we consider them as test oracles who provide expected behavior based on their numeric model of the program. However, the basis for generating such a range of values (i.e., expected intervals) will vary depending on the task and the level of expertise of the user. The user may specify the expected interval based on the values of cells which will be referenced in the formula (i.e., based on the arithmetic model the user may have), or knowledge in the application domain, or experience on the same or similar task handled earlier.

The usual spreadsheet formula evaluation results in a discrete value d (see Figure 6.1). Corresponding to each formula cell for which an expected interval is attached, interval computation is performed using interval program. The execution of the interval-program is based on an interval arithmetic semantic of the operators used in the various formulas. This leads to the computation of a bounding interval B for the desired formula. The purpose of a computed interval is just to control the user's expectation so that the expected interval lies within a range of reasonable values. The reasonableness of the expected interval provided by the user is verified by the system using interval analysis performed behind. Once the necessary values are available from the three sources, comparison between the values d, E and B is performed. The comparison $d \in E \wedge E \subseteq B$ reports no symptom of fault whereas $d \notin E \vee E \nsubseteq B$ indicates a symptom of fault. For those formula cells for which a symptom of fault is generated, further investigation is carried out using a fault tracing strategy.

The fault tracing strategy encourages spreadsheet-based verification. Spreadsheet-based verification is performed after the necessary formulas are defined and the corresponding expected intervals are attached. This corresponds to a testing phase which takes place after development is completed. In order to trace propagated faults, the fault tracing strategy provides a mechanism of identifying the most influential faulty cell corresponding to a formula cell in which a symptom of fault is detected. On the other hand, cell-based verification is performed for each formula cell. This is usually performed while developing a spreadsheet. In such cases, the user is interested in verifying each desired formula cell at a time after providing the necessary formula. This could be a time consuming process although it enables the user to fix the faults before they are propagated to other dependent formula cells.

6.4 Spreadsheet Interval Computation

One of the information source, the interval-program, used in the spreadsheet formula test process (see figure 6.1) requires attaching intervals and performing interval analysis on spreadsheets. The idea of interval computation on spreadsheets was introduced for the purpose of dealing with imprecise data and numerical constraints by Hyvönen and Pascale [45]. Instead of exact numbers, users can perform computations on intervals using spreadsheets. The way interval computation is performed in interval-based testing is different from the interval computation used in [45]. In [45], interval computation is performed by designing interval spreadsheet functions. That is for each worksheet function, a corresponding interval function is defined. It performs its computations based on interval arguments. For example, for the SUM function, an equivalent interval function ISUM is defined to compute the sum of interval numbers. This is possible because spreadsheet systems (e.g., MS Excel) allow user-defined functions to be used for computations. The interval spreadsheet functions defined are generalizations of ordinary spreadsheet functions as intervals are extensions of real numbers.

However, defining interval spreadsheet functions is not convenient in interval-based testing. First, it requires the user to learn new functions designed to perform interval computation. This would incur additional cognitive load to the user. The goal of interval-based testing is to perform interval computation from the ordinary spreadsheet formulas defined by the user thereby allowing the user to work in the usual way of spreadsheet development. Moreover, values of cells during interval computation in interval-based testing could be from two sources (in the case of mixed arithmetic). If a formula involves cells of which some have intervals attached and others without intervals attached, then the corresponding values are read from expected spreadsheet and ordinary spreadsheet respectively. **Ordinary spreadsheet (O_s)** is the usual spreadsheet in which computation is based on discrete values. **Expected spreadsheet (E_s)** is a behind-the-scene spreadsheet which contains input intervals and expected intervals for input and formula cells respectively. To avoid referring to two spreadsheets, those discrete values can be propagated to the expected spreadsheet. This requires updating the expected spreadsheet whenever changes are made to the discrete values in the ordinary spreadsheet. As a result we have chosen to perform interval computation without requiring the user to learn new functions. During the verification process, each formula is parsed and then those cell references involved in the formulas are assigned their corresponding intervals. Then the interval module is called to perform interval computations.

In interval-based testing, in addition to discrete values, interval values are attached to the desired input and formula cells during the development of an ordinary spreadsheet. For numeric input cells, the user specifies the range of reasonable values in the form of intervals which serve as input domains. For formula cells (cells which are intended for numerical computation), the user specifies the expected outcome of the formula again in the form of intervals. The attached intervals will be stored as strings (since the spreadsheet system neither support interval data types nor allow user defined data types) in a behind-the-scene spreadsheet(E_s) using the same cell coordinates as the cells in the ordinary spreadsheet. During interval computation, the interval strings will be converted into interval data types. For a formula defined at the ordinary spreadsheet (O_s), the formula is evaluated based on the interval values stored in E_s and the resulting interval will be stored as an interval string in the respective cell in the bounding spreadsheet (B_s). The **bounding spreadsheet (B_s)** which is a behind-the-scene spreadsheet contains computed bounding interval values.

For some of the input cells, the user may not attach input intervals. These cells could contain conceptually constant values or the user doesn't want to attach intervals. Similarly, the user may not attach expected intervals for some formula cells. These cells are not intended for verification and hence no interval computation is performed. For the interval computation of a formula which references input cells and formula cells which do not have intervals attached, the corresponding discrete values from the ordinary spreadsheet are used. The intervals in this situation will be of width zero.

6.4.1 Input Intervals

Intervals attached to input cells are domains for those cells out of which values are used for ordinary spreadsheet computations. Input intervals can be considered as narrowed sets (restricted domain) containing possible valid test cases of the cells to which they are attached. The specification of an interval is then an implicit specification of valid test cases. If a formula is validated as correct based on such intervals, then it can be considered as valid for those test cases. Narrowing the possible values of test cases to a reasonable range of values (i.e., intervals) enables us to judge the validity of a formula cell based on such an interval rather than validating it for each test case. Therefore, in essence we are reducing the iterative process of judging the validity of a formula cell for each test case to a single validation activity based on interval analysis. This also saves time by doing validation only once for a formula which may take a variety of values if different test cases are applied.

Cells in an ordinary spreadsheet and in the corresponding expected and bounding spreadsheets are related by their cell coordinates. The definition of relation is given below.

Definition 6.4 (Relation) Two non-empty cells $C(i, j) \in O_s$ (a cell in an ordinary spreadsheet) and $E(m, n) \in E_s$ (a cell in an expected spreadsheet) are said to be related iff $i = m$ and $j = n$. Similarly, this relation is extended for cells containing computed bounding intervals $B(p, q) \in B_s$ (cells in a bounding spreadsheet). The extended relation requires that $i = m = p$ and $j = n = q$.

For a given cell in an ordinary spreadsheet with coordinate (i, j), let $E(i, j)$ be its related cell in the expected spreadsheet. Algorithm 1 describes the necessary steps to attach an interval to a single cell or group of cells.

Intervals are supposed to be attached only to numeric cells. If an attempt is made to attach an interval to cells of another types, an error message is provided as intervals are meaningless for other types of cells. In some cases there may be a need to attach

Algorithm 1 Algorithm to attach an interval to a group of cells

get the coordinates of the first and last cells in O_s
for $i = RowFirstCell$ to $RowLastCell$ **do**
  for $j = ColumnFirstCell$ to $ColumnLastCell$ **do**
    $E(i, j) \leftarrow interval$
  end for
end for

the same interval to a group of cells which have the same functionality and similar values. Algorithm 1 describes the necessary steps to attach an interval to a group of cells. This algorithm can be also used to attach an interval to a single cell when the first and last cells are the same.

The user is working only at the ordinary spreadsheet and (s)he is not aware of the associated spreadsheets containing expected intervals and computed bounding intervals. The expected spreadsheet is created at the moment the user started to attach an interval whereas the bounding spreadsheet is created when interval computation is performed. The association between the ordinary spreadsheet and the expected spreadsheet (also bounding spreadsheet) is a name association. In other words, expected spreadsheet and bounding spreadsheet are given the same name as the ordinary spreadsheet but with different extensions.

6.4.2 Expected Intervals

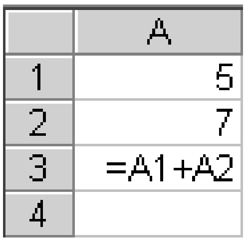
It is legitimate to assume that a user sets a goal first whenever (s)he wants to perform some calculation. Corresponding to the goal set, (s)he will have some expectation of the outcome of the computation. Therefore, the user is expected to specify the expected interval of a formula before carrying out the computation. Expected intervals represent a range of acceptable (or reasonable) values so that if a computed value lies within that range then the associated formula is considered as correct.

In a typical spreadsheet system, we can classify the types of formulas used as conditional and non-conditional. The non-conditional formulas are common formulas such as adding values of groups of cells whereas conditional formulas perform computation based on a condition. Conditional formulas are those which involve the IF function. For non-conditional formula cells, users attach a single interval to describe the expected outcome. However, for conditional formula cells, the expected outcome depends on the decision taken and hence on the branch actually executed. Therefore, for conditional formulas, the user is expected to attach intervals corresponding to each branch of the decision that will be executed. For example, for the formula $IF(A1 > 5, A1 * B1, A1 - B1)$, the expectations are different based on whether the comparison $A1 > 5$ is true or false. The true branch requires an expectation corresponding to the product $A1 * B1$ (i.e.,the then part) and the false branch requires an expectation corresponding to the difference $A1 - B1$ (i.e., the else part).

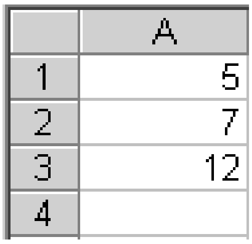
For nested IFs the procedure is the same except that the representation becomes a set of intervals. To generalize the number of expected intervals which are needed for nested IFs, let N be the number of nested IFs used in a formula. Then, $N + 1$ expected intervals are required. However, for a formula which involves N IFs without being nested, we need 2^N expected intervals (e.g., IF(...) + IF(...)). The algorithm to attach expected intervals is the same as in for input intervals except that multiple intervals are attached to a single cell as a set of intervals for a conditional formula. However, nested and multiple IFs are less likely to be used by a significant number of users.

6.4.3 Computation of Bounding Intervals

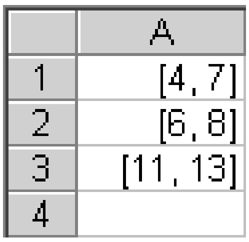
A bounding interval is a computed interval used to check the reasonableness of the expected interval specified by the user. The computation of bounding intervals is based on the formulas defined at the ordinary spreadsheet. During interval computation, the cells referenced in the formula assume interval values and the operators are used in interval arithmetic semantics. As in the case of attaching intervals, the procedure for the computation of bounding intervals for conditional and non-conditional formula cells is different. For non-conditional formula cells, the computation is straightforward; instead of the usual discrete values of cells, the corresponding input and expected intervals from the expected spreadsheet are used. In addition, arithmetic operators are used in the interval arithmetic context. If there are cells for which no interval is attached, the values from the ordinary spreadsheet are used. For example, figure 6.2(a) and figure 6.2(b) show a simple spreadsheet to compute the sum of two numbers and figure 6.2(c) shows the corresponding input and expected intervals. Figure 6.2(d) shows the bounding interval for the formula cell A3.



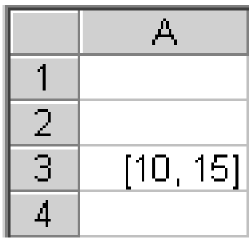
(a) Formula view



(b) Normal view



(c) Input & expected interval

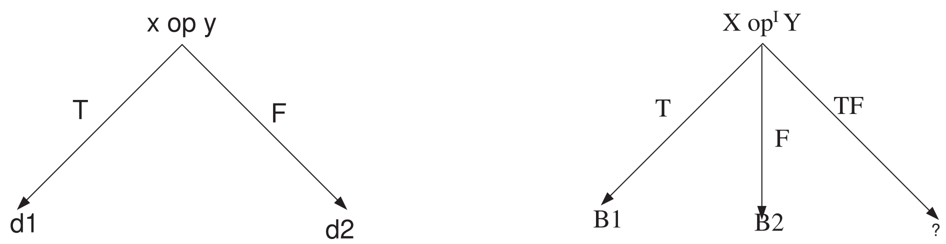


(d) Bounding interval

Figure 6.2: Computation of a bounding interval for a non-conditional formula

Computation of Bounding Intervals for Conditional Formulas

The computation of a bounding interval for a conditional formula requires careful examination. The first issue is the determination of the branch to be executed. As the comparison operators used in conditions are interval comparison operators, the decision is generally not a binary decision. This is due to the fact that the set of intervals are not totally ordered. Therefore, we need some mechanism of transforming the three-valued decision resulting from interval comparison (see section 5.3) into a binary decision. Moreover, the decision taken at the interval-spreadsheet level should also conform to the decision taken at the ordinary spreadsheet level. If the intervals under consideration are disjoint, then the comparison at the interval spreadsheet level conforms to the comparison at the ordinary spreadsheet level. However, if the intervals are non-disjoint, then the comparison at the interval spreadsheet level becomes undecidable (see figure 6.3(b)). Let $X = [a, b], x \in X$ and $Y = [e, f], y \in Y$. Consider xopy and $X op^I Y$ where $op \in \{<, \leq, >, \geq, =, \neq\}$. Figure 6.3 shows the possible decisions that can be made by the spreadsheet-program and by the interval-program.



(a) spreadsheet-program (b) interval-program

Figure 6.3: Comparisons in spreadsheet-program and interval-program

Recall that in chapter 5, we have seen that different approaches were used to transform interval comparison into Boolean comparison which are only approximations (or in some cases simple choices) that do not exactly provide the same decision. Therefore, to maintain the consistency of the decision taken by an ordinary spreadsheet and interval spreadsheet, we can consider the following cases.

1. Interval Comparison

   In this option, we simply use interval comparison operators to determine the branch which will be executed. To illustrate the situation, consider the example in figure 6.4.

	A	B	C
1	5	8	=IF(A1 > B1,A1*B1,A1-B1)
2			
3			

(a) ordinary

	A	B
1	[5, 8]	[6, 9]
2		
3		

(b) expected

Figure 6.4: A spreadsheet with conditional cell

Figure 6.4 depicts an ordinary spreadsheet which contains a conditional formula. Even though this spreadsheet displays the formula instead of the computed value, we can observe that the branch executed is the else part. But the comparison at the interval spreadsheet level is undecidable (i.e., the value is TF). Therefore, in this situation we have to compute values corresponding to the two branches.

As we have seen in chapter 5 section 5.3, a better approach for interval comparison is the one used in FORTRAN 95 [66]. However, this approach requires the programmer to specify the type of comparison (i.e., Set theoretic, Certainly, Possibly). In interval-based testing, the user is working only at the ordinary spreadsheet level and (s)he is even not aware whether such comparisons are performed. Therefore, interval comparison in this form is not applicable for determining the branch which will be executed. What we can do is to perform the comparison based on either the Set theoretic or the Certainly or the Possibly context. To do so, we shall identify the type of comparison which conforms to Boolean comparison.

Table 6.1 presents the truth values of the conditional formula defined in figure 6.4. From this table, we can see that Set theoretic interval comparison conforms to Boolean comparison (see the rules for the interval relational operators in chapter 5 section 5.3).

Now let us consider another example to see whether this conformance of interval

Operator	**Boolean Comparison**	**Interval Comparison**		
		Set theoretic	Certainly	Possibly
$=$	F	F	F	T
$\geq$	F	F	F	T
$>$	F	F	F	T
$\leq$	T	T	F	T
$<$	T	T	F	T
$\neq$	T	T	F	T

Table 6.1: Comparing Boolean comparison and Set theoretic comparison

comparison (Set theoretic) and Boolean comparison holds in other situations. Let $A1 = 5$ and $B1 = 6$ be cell values in the ordinary spreadsheet and $A1 = [4, 7]$ and $B1 = [4, 7]$ be the corresponding input intervals. As we can see from table 6.2, the conformance between Set theoretic interval comparison and Boolean comparison does not always hold. Of course, we can see that the Boolean comparison conforms to none of the interval comparisons. Hence, if these interval comparisons are used in interval-based testing, the testing system will produce false symptoms of faults since ordinary spreadsheet and interval spreadsheet decisions will take different branches.

Operator	**Boolean Comparison**	**Interval Comparison**		
		Set theoretic	Certainly	Possibly
$=$	F	T	F	T
$\geq$	F	T	F	T
$>$	F	F	F	T
$\leq$	T	T	F	T
$<$	T	F	F	T
$\neq$	T	F	F	T

Table 6.2: Comparing Boolean comparison and interval comparisons

2. Interval Partitioning

   The other possibility is to partition each interval under comparison into a disjoint union of intervals. Let X and Y be non-empty intervals and $X \cap Y \neq \emptyset$. X can be partitioned into $X' = X \setminus X \cap Y$ and $X \cap Y$. Similarly, Y can be partitioned into $Y' = Y \setminus X \cap Y$ and $X \cap Y$. Table 6.3 shows the possible combinations of comparisons. We can see that interval comparisons of all combinations except the last row conform to Boolean comparisons based on single interval comparison defined in chapter 5 section 5.3. The interval comparison of the last combination is the same as the comparison in table 6.2 which does not conform to Boolean comparison.

X	**Y**
X'	Y'
X'	$X \cap Y$
$X \cap Y$	Y'
$X \cap Y$	$X \cap Y$

   Table 6.3: Interval partitioning

   Note that in some cases, either X' or Y' can be empty set (i.e., $X \subseteq Y$ or $Y \subseteq X$). In this case too, the comparisons follow the same procedure.

3. Decision Propagation from Ordinary Spreadsheets

   As we have seen from the above two options, it is not always possible to maintain the consistency between ordinary spreadsheet decision and interval spreadsheet decision by using Boolean comparison at the ordinary spreadsheet and interval comparison at the interval spreadsheet. Therefore, we should devise a mechanism to propagate ordinary spreadsheet decisions to the interval spreadsheet so that the same decision and hence the same branch is executed. To do so, we have to attach an ordinary spreadsheet pointer for the cells involved in the conditional part of an IF function when traversing the abstract syntax tree. In other words, during the evaluation of the IF function, the values used for the comparison are read from the ordinary spreadsheet. This can be accomplished whenever necessary since related cells have the same coordinates.

Algorithm 2 Interval computation for a non-conditional formula cell

if expected spreadsheet exists **then**
  Initialize bounding spreadsheet
  for each formula cell in the selected region **do**
    if formula has an attached expected interval **then**
      1. get the coordinate of the cell
      2. parse formula
      3. convert the abstract syntax tree into postfix form
      4. perform interval arithmetic
      5. $B(i,j) \leftarrow interval$ where $B(i,j)$ is a cell in the bounding spreadsheet with the coordinate read in step 1
    end if
  end for
end if

Algorithms 2 and 3 describe the general procedures to perform interval computation for a non-conditional formula and conditional formula respectively.

Bounding Interval for a Formula Referring to a Conditional Cell

The other issue is how to compute the bounding interval for a formula cell referencing a conditional cell. If a formula refers to a cell which contains an IF function, then the issue is which expected interval of the conditional cell should be used in the computation. There are three alternatives.

1. Using all the expected intervals. In this alternative, all the expected intervals attached to a referenced cell are used during the computation of the bounding interval for the referencing cell. This, however, involves extra intervals and hence results in a wide bounding interval.

Algorithm 3 Interval computation for a conditional formula cell

if expected spreadsheet exists **then**
  initialize bounding spreadsheet
  for each formula cell in the selected region **do**
    if formula has an attached expected interval **then**
      1. get the coordinate of the cell
      2. Parse formula
      if formula is an IF function **then**
        a. get the cell address(es) in the conditional part
        b. get values of those cells with addresses as in (a) from O_s
        c. determine branch to be executed
      end if
      3. Convert the desired part of the abstract syntax tree into postfix form
      4. Perform interval arithmetic
      5. $B(i, j) \leftarrow interval$ where $B(i, j)$ is a cell in the bounding spreadsheet with the coordinate read in step 1
    end if
  end for
end if

2. Using bounding intervals. Computation based on the bounding intervals of referenced cells follows the usual way of spreadsheet computation. In principle, computation of bounding intervals based on the data dependency graph (DDG) is appropriate. However, similar to the first alternative, computation of bounding intervals based on the DDG results in too much width as all precedent cells' bounding intervals are used for the computation.

3. Using the expected interval corresponding to the branch executed. The above alternatives introduce extra width to the computed interval. Therefore, an appropriate solution is to be able to use the expected interval corresponding to the branch executed. This alternative presupposes the ability to identify the branch executed. However, the identification of which of the expected intervals corresponds to the branch executed for the current evaluation is a difficult task. The information regarding the branch executed is a dynamic information which is not available but after the evaluation of the formula. Nevertheless, if the branch information is stored during the computation of the bounding interval of a cell, this information can be used during the computation of bounding intervals for those cells which reference the cell. When reading expected intervals of referenced cells, if a cell has a set of intervals attached, then the bounding spreadsheet can be consulted to get branch information. Based on the branch information, the respective expected interval is used during the computation.

Computation Following MS Excel Style

The prototype is implemented on top of MS Excel. The choice of this spreadsheet system is just due to its wide availability. To facilitate a careless style of computation, MS Excel allows a user to mix-up text data and numeric data in a computation without signaling any error message (except division where the divisor is text or an empty cell). In such situations, text data is treated as zero value. Empty cells are also treated in a similar manner. Therefore, to maintain the same style of computation at the ordinary spreadsheet (MS Excel computation) and bounding spreadsheet (interval computation), text and empty cells are treated as zero values during interval computation. Values in the ordinary spreadsheet are used during interval computation if the cells referenced in a formula have no intervals attached (i.e., for a cell $C(i,j)$ referenced in a formula, $E(i,j)$ is empty).

6.5 Detecting Existence of Faults

Generally, the main task in testing a program is to be able to detect the existence of faults in the program. To achieve this, different techniques can be used. As mentioned earlier, for the purpose of testing spreadsheets, a user specifies intervals for those input cells which may assume different values. Those cells which do not assume different values can be represented by an interval of width zero. In addition, for a formula cell which needs to be tested, the user specifies the expected outcome of the computation in the form of an interval. Therefore, for a formula cell under test, there are two values to be computed: a value computed by the spreadsheet program (**d**) (see Figure 6.1) based on the values of the cells referenced in the formula and a bounding interval (**B**) computed by interval program based on interval arithmetic using the interval values of those referenced cells. The interval program can be viewed as an equivalent of a spreadsheet program where the values of cells are represented as intervals and the computation is performed based on interval arithmetic. In order to infer the existence of a symptom fault in a formula cell, the three values **d, E,** and **B** (see Figure 6.1) which are generated by different sources should be compared.

In section 6.4.3, we have seen that non-conditional and conditional formulas require different considerations when computing a bounding interval. Similarly, the comparisons between **d, E,** and **B** need different considerations for non-conditional and conditional formulas to report any symptoms of faults.

6.5.1 Comparator for Non-conditional Cells

If a formula cell does not contain an IF function, then the comparison is straightforward. Let $E = [e_1, e_2]$ and $B = [b_1, b_2]$ be the expected and bounding intervals respectively for a given formula cell. Let d be the discrete value of the given formula. Under normal conditions, $\mathbf{d} \in \mathbf{B}$ since d represents the value of a real-valued function and B represents the value of the interval extension of the real-valued function (see section 5.2.3). For the comparison of d, E, and B, there are two cases to consider.

Case 1: $\mathbf{d} \in \mathbf{E}$ and $\mathbf{E} \subseteq \mathbf{B}$

As the computed interval value of a formula is bounded by the minimum and maximum values of the possible computation (this is by definition of interval arithmetic), the expected interval should lie within the computed interval. In addition, the value computed by the spreadsheet program should lie within the expected range of computation. Hence, in this case, we can say that there is no symptom of fault. In terms of the end points of the intervals, the comparisons can be defined in the following way. Figure 6.5 shows the situation in which no symptom of fault is detected in a formula.

1. $d \in E$ iff $e_1 \leq d \leq e_2$
2. $E \subseteq B$ iff $b_1 \leq e_1 \wedge e_2 \leq b_2$

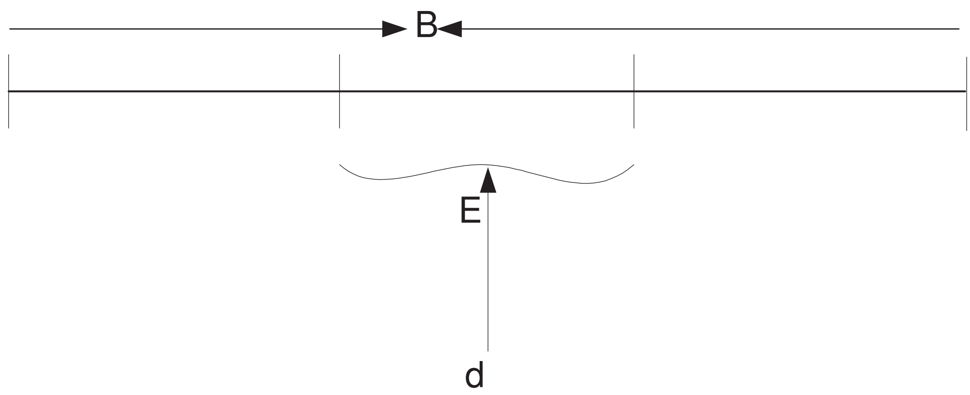


Figure 6.5: Conformance between **d, E** and **B**

Case 2: d $\notin$ E or E $\nsubseteq$ B

In this case, there is an indication of a symptom of fault. The fault may be in the formula or in the user's perception of expected results. Of course, testing is performed based on the assumption that there is a correct behavior of a program against which the actual result is compared. However, we can not always take for granted that the expected behavior is correct. When E is specified incorrectly, the fault signal serves as a reminder to revise the perceived range of values. In the situation where $d \notin E$ due to some faults in the formula, the actual result is shifted from the expected result. In the second possibility where $E \nsubseteq B$, faults affect the bounding interval computed for the formula and create a discrepancy between E and B. There are different scenarios describing the situation where there is a symptom of fault (see figures 6.6, 6.7, and 6.8).

Scenario 1: $d \notin E$ **and** $E \subseteq B$ $(i.e., \forall x \in E$ **and** $d \in B, d < x$ **or** $d > x)$

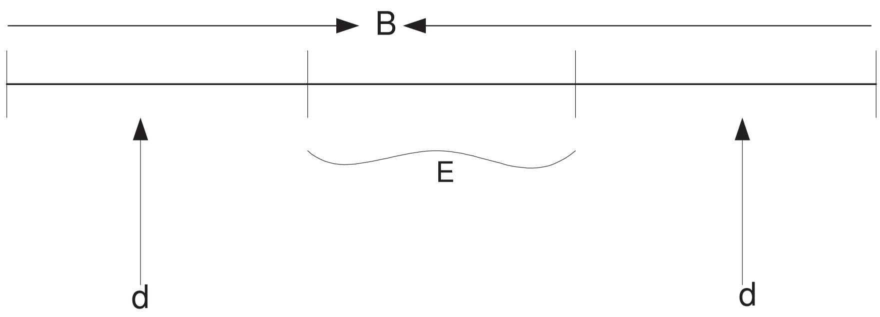
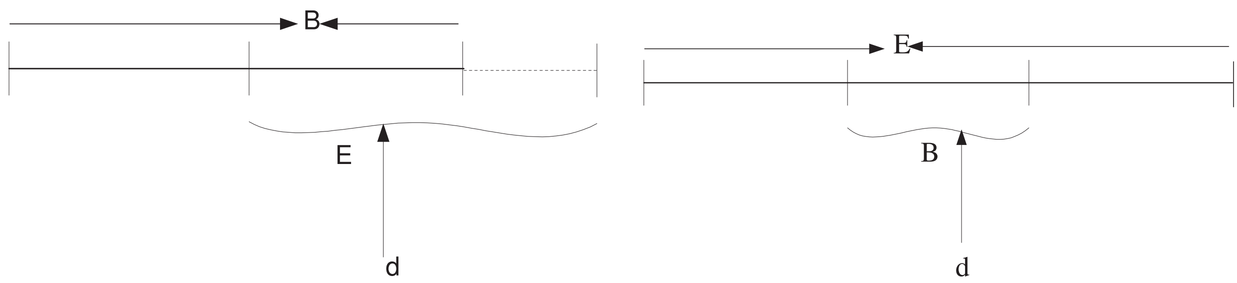


Figure 6.6: Discrepancy between **d** and **E**

Scenario 2: $d \in E$ **and** $E \nsubseteq B$ **(with** $E \cap B \neq \emptyset$**)**



(a) (b)

Figure 6.7: Discrepancy between **E** and **B**

Scenario 3: $d \notin E$ **and** $E \nsubseteq B$

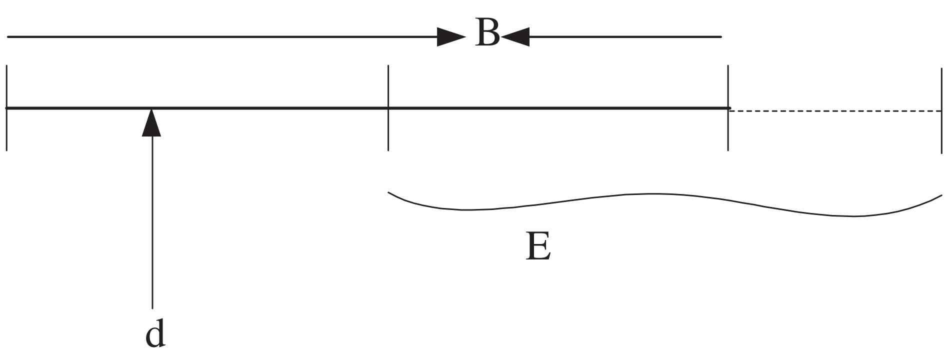


Figure 6.8: Discrepancy between **d**, **E**, and **B**

In all the scenarios $d \in B$. This is based on the assumption that all referenced cells have discrete values lying within the corresponding intervals specified. In other words, treating ordinary spreadsheet formulas as ordinary functions and the corresponding interval computations as interval extensions, we see that ordinary spreadsheet functions' values belong to their interval extensions (see 5.2.3). The algorithm for detecting a symptom of fault in a non-conditional formula is given in algorithm 4. Note that a cell $C(i,j)$ represents a cell in the ordinary spreadsheet with coordinate (i,j); $d(i,j)$ represents the discrete value of the cell C; $E(i,j)$ represents a cell for the attached interval in the expected spreadsheet; $B(i,j)$ represents a cell for the computed interval in the bounding spreadsheet; R represents the selected region for verification; F is the set of formula cells in R.

Algorithm 4 Comparator for non-conditional formula cells

```
for each numeric cell C(i,j) ∈ R do
  if E(i,j) ≠ ∅ then
    if d(i,j) ∉ E(i,j) then
      mark C(i,j) incorrect
    else
      if C(i,j) ∈ F then
        if E(i,j) ⊈ B(i,j) then
          mark C(i,j) incorrect
        else
          mark C(i,j) correct
        end if
      end if
    end if
  end if
end for
```

Example 6.5: Detecting symptoms of faults

A user is interested to know how much (s)he will get by depositing different amounts of money with different rates. Depending on the amount of money deposited, the in-

terest rate varies. The task in the spreadsheet shown in figure 6.9 (taken from [21]) is to calculate the interest for each amount deposited (product of column A entries by column B entries row-wise) and finally to sum up the overall total that could be gained from the deposit. To carry out the testing process, the user documents the input and the expected range of values as shown in figure 6.11. Interest rates are fixed or constant values and as a result the user does not attach intervals for the rate column. However, since the rates are for a ceratin range of values, the user specifies ranges of values for the deposits in column A. Based on the formulas used in the ordinary spreadsheet in figure 6.10, an interval computation is performed resulting in the bounding intervals shown in figure 6.12. Finally, the testing system verifies the given spreadsheet and reports a symptom of fault for some of the cells. Those cells with symptoms of faults (see figure 6.13) are marked red (dark in this paper) and those without a symptom of fault are marked yellow (grey in this paper). In addition, those formula cells which are not verified are shown with their default white color. As the figure shows, a wrong reference is entered for the formula in cell C4 and even copied to C5.

	A	B	C
1	deposit	rate	payment
2	3000	0.03	90
3	5000	0.035	175
4	8000	0.05	280
5	10000	0.06	500
6			1045
7			

Figure 6.9: A spreadsheet to compute interest

	A	B	C
1	deposit	rate	payment
2	3000	0.03	=A2*B2
3	5000	0.035	=A3*B3
4	8000	0.05	=A4*B3
5	10000	0.06	=A5*B4
6			=SUM(C2:C5)
7			

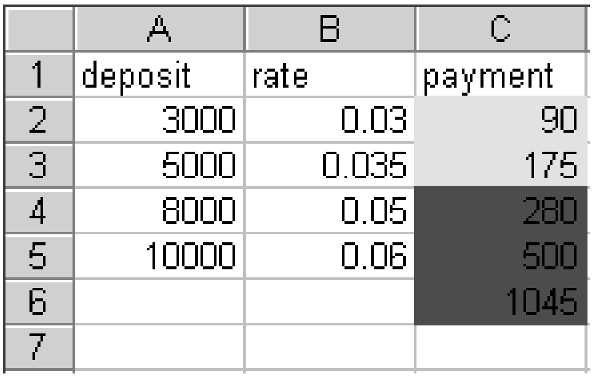
Figure 6.10: Formula view of the interest spreadsheet

	A	B	C
1			
2	[2750:3100]		[87:93]
3			[150:200]
4	[7890:8000]		[400: 400]
5	[9800:11000]		[600: 666]
6			[1400:1500]
7			

Figure 6.11: Expected spreadsheet

	A	B	C
1			
2			[82.5:93]
3			[175:175]
4			[276.15:280]
5			[490:550]
6			[1237:1359]
7			

Figure 6.12: Bounding spreadsheet



	A	B	C
1	deposit	rate	payment
2	3000	0.03	90
3	5000	0.035	175
4	8000	0.05	280
5	10000	0.06	500
6			1045
7			

Figure 6.13: A spreadsheet with symptom of faults

As we can see from figure 6.13, the three formula cells have symptoms of faults. The expected intervals were specified based on the goal of row-wise computation. Cells C4 and C5 have wrong references in their formulas. The formula specified for cell C6 which contains the total sum of the payment has no fault within the formula. Nevertheless, since some of the cells which are referenced in the formula are faulty, it turns out to be also faulty. In addition, we can see that the expected interval specified for this cell does not contain the correct value of the cell. Therefore, fixing the faults in the preceding cells cannot render this cell to be correct. This indicates that expected intervals may be specified incorrectly due to different reasons (e.g., lack of domain knowledge or incorrect goal assumption). In such situations, expected intervals which correspond to the goal of the computation should be revised. Actually, the first thing to examine is the expected interval and the formula itself before trying to locate possible propagated faults.

Another discrepancy that we can observe from the above spreadsheet is the violation of the assumption that $\mathbf{d} \in \mathbf{B}$ for the cell C6. This is due to the fact that the discrete values of some of the cells referenced in the formula of C6 do not lie in their corresponding expected intervals. For example, the value of cell c4 is 280 while its expected interval is [400:400].

During the process of fault detection, a special case that needs to be considered is when the bounding interval is of width zero. For example, in figure 6.12 the bounding interval for cell C3 is 175 which is represented as [175:175]. Since the referenced cells do not have intervals attached, the bounding interval is the discrete value computed in the ordinary spreadsheet. However, an expected interval is attached for this cell (see figure 6.11). In this case, the usual way of comparison $E \subseteq B$ reports a symptom of fault which is wrong. Therefore, if the bounding interval of a given formula cell is of width zero, then only the comparison $d \in E$ is performed. Other special considerations of the comparisons between **d**, **E**, and **B** are given in section 6.5.3.

6.5.2 Comparator for Conditional Cells

If an IF function is involved in a formula, then the formula will have more than one expected interval. For the evaluation of the decision to compute the bounding interval, the decision is propagated from the ordinary spreadsheet. As a result there is only one bounding interval corresponding to the branch executed. Let $E = \{E_1, E_2, ..., E_n\}$ represent the expected intervals specified for a conditional cell and $B = B_i$ represent the bounding interval corresponding to branch i. In this case, the expected interval corresponding to the branch executed is compared with the bounding interval as given below.

1. $d \in E_i$ where i represents the branch executed and

2. $E_i \subseteq B_i = B$

Had it not been for the propagation of the decision from the ordinary spreadsheet, we should have considered the computation of bounding intervals corresponding to each branch of the decision if the condition is undecidable. That is, for $B = \{B_1, B_2, ..., B_m\}$ representing the bounding intervals, the comparison would have been as follows.

1. $d \in E$ iff $\exists E_i \in E$ such that $d \in E_i$

2. $E \subseteq B$ iff $\forall E_i \in E, \exists B_j \in B$ such that $E_i \subseteq B_j$

We can observe how complicated and ineffective this option is. Similar to the propagation of decision from ordinary spreadsheet, we need to find a way to get the branch information for the current execution (i.e., which branch is executed). One way to do so is to store branch information during interval computation. Therefore, information regarding the branch executed is stored in the bounding spreadsheet during interval computation. In order to carry out the comparison for a conditional formula cell, the branch information is extracted from the bounding spreadsheet in the corresponding cell.

The algorithm for the comparison of conditional formula cells is the same as the algorithm for non-conditional formula cells except that the branch information should be extracted from the bounding spreadsheet before any comparisons between **d**, **E**, and **B** are made. Algorithm 5 describes a comparator for conditional formula cells.

Algorithm 5 Comparator for conditional formula cells

```
for each numeric cell C(i,j) ∈ R do
  if E(i,j) ≠ ∅ then
    get branch information from B_s
    let k be the branch executed
    if d(i,j) ∉ E_k(i,j) then
      mark C(i,j) incorrect
    else
      if C(i,j) ∈ F then
        if E_k(i,j) ⊈ B(i,j) then
          mark C(i,j) incorrect
        else
          mark C(i,j) correct
        end if
      end if
    end if
  end if
end for
```

6.5.3 Special Case of the Comparator

As we have seen in section 6.5, the basic comparisons for detecting the existence of a fault in a formula are based on the values **d**, **E**, and **B**. The case in section 6.5.1 figure 6.7 (i.e., $d \in E \cap B$, but $E \nsubseteq B$) motivates some further consideration to generate symptom of faults. In some situations, it may be difficult to imagine that the user's expectation lies completely within the bounding interval computed by the testing system even though there is no fault in the formula. As a result, it is worth of considering to relax the comparison $E \subseteq B$ so that a symptom of fault is generated only when a certain percentage of the user's expectation is not met. The argument here is that if the expectation of the user is met to a large extent, then the likelihood of a fault in the formula is minimal. Similarly, if the user's expectation is not met to a large extent, then the likelihood of a fault in a formula is high and hence a symptom of fault should be provided. However, the question is that how much percentage of the user's expectation should be met to say that there is no symptom of fault. This can be determined only after carrying out experiments and observing users practices. But, the basic comparison to be investigated is between the width of the intervals $E \cap B$ and $E \setminus E \cap B$. The width of an interval [a, b] is computed as b - a. First, we consider the cases where symptoms of faults are generated.

1. $w(E \setminus E \cap B) = w(E)$ (i.e., $E \cap B = \emptyset$). This indicates that the user's expectation and the computed interval do not agree at all.

2. $w(E \setminus E \cap B) < w(E \cap B)$, but $d \notin E \cap B$. This indicates that either $d \notin E$ or $d \notin B$ or both which is an erroneous situation.

3. $w(E \cap B) < w(E \setminus E \cap B)$. This indicates that the larger part of the expectation is not met and hence a symptom of fault is generated.

On the other hand, the following cases indicate no symptom of fault (item 1) though some of the cases require experimental investigation (cases in item 2).

1. $w(E \setminus E \cap B) = 0$ (i.e., $d \in E$ and $E \subseteq B$). In this situation, the expectation is fully satisfied.

2. $w(E \cap B) > w(E \setminus E \cap B)$. This indicates that E is to a large extent a subset of B (i.e., the intersection lies to a large extent in E) which shows that the expectation is met to a larger extent. In other words, to a large extent E and B agree. Hence the likelihood of having a fault in the formula is less. This can happen when most or the majority of the cells involved in the formula do not have intervals attached. In principle, if intervals are not attached to most of the cells involved in the formula, then the expected interval for the given formula should not be wide. However, we don't expect the user to have such sharp estimation of the expected outcome of the computation. Further comparisons to investigate are the effect of fault detection when the size of the intersection between E and B varies. The following values can be experimentally investigated.

   - $w(E \cap B) \geq (\frac{1}{2}) * w(E)$. In this situation, more than 50% of the expectation is met.
   - $w(E \cap B) \geq (\frac{3}{4}) * w(E)$. In this situation, more than 75% of the expectation is met.

There are some factors which affect the comparison between E and B (i.e., situations where B will be a weak controlling mechanism). Among them are dependency problems which result in too much width in the computation of B (the width becomes large rapidly). The other is division that involves intervals containing the value zero which often results in infinite end points.

6.6 Spreadsheet Verification Coverage

For a spreadsheet program to be correct, every formula as well as every input value should be correct. Users are more likely to make errors while defining formulas than when entering input values because formulas may consist of cell references, operators, and functions to carry out computations. As a result, verification information is mainly needed for formula cells. To get a general verification information, one needs to examine each cell value which is a tedious task. However, if a testing system is employed, such an information can be provided by the testing system.

In interval-based testing, after verification is performed for a selected region of cells, those cells which have symptoms of faults and others which do not have symptoms of faults are marked with different colors. From this visual information, a user can identify those which have symptoms of faults and need further investigation. Besides, it may be also helpful to know to what extent a given spreadsheet is verified. The question that can be raised is "How do we know the correctness of a given spreadsheet?". Correctness information can be inferred from the number of formula cells which are verified and correct. In a given spreadsheet, we should see a difference in the extent of verification when the number of formula cells which are verified and correct increases. Therefore, information similar to "test coverage" in conventional software testing can be provided to indicate the extent of formula verification. The verification information can be inferred from the number of formula cells which are verified and correct with respect to the total number of formula cells. It can be computed in the following way.

Spreadsheet Verification Coverage (SVC) = $\frac{F_v}{F_t}$ where F_v represents the number of verified numeric formula cells and F_t represents the total number of numeric formula cells in a spreadsheet. The spreadsheet verification information about the percentage of verified cells will initiate the user to verify the remaining unchecked numeric formula cells. To visualize this, the testing system provides a different color to checked, unchecked, and faulty formula cells.

6.7 An Example

Consider the house rent calculation problem where the task is to calculate the total income from the rent of different types of houses with different sizes and rental rates. The original spreadsheet on which the user is working on is shown in figure 6.14. In order to test this spreadsheet, the user attaches intervals to some of the desired input and formula cells. Figure 6.16 shows the intervals attached by the user and documented in a behind-the-scene spreadsheet, the expected spreadsheet. During the verification of the spreadsheet, interval computation is carried out for those formula cells for which expected intervals are attached. The computed intervals are again documented in a behind-the-scene spreadsheet as shown in figure 6.17. Finally, the comparator compares the values of those cells from the three sources (figures 6.14, 6.16, and 6.17) which are selected for verification and marks those cells which have symptoms of faults and those which are correct as shown in figure 6.18.

As figure 6.18 shows, there are a lot of cells marked with red (dark in this paper) which have symptoms of faults. Those which do not have symptoms of faults are marked yellow (light grey in this paper). However, this spreadsheet is believed to be fault-free by the user. There are two reasons for the discrepancies among the three values of each formula cell marked faulty. First, if the expected interval does not contain the discrete value computed by the spreadsheet, then the error is in the specification of the expected interval (since this particular spreadsheet is fault-free). This could be due to either lack of domain knowledge or misunderstanding of the problem. Therefore, such discrepancies may serve to revise the goal and to have a better understanding of the problem. The cells **F11, F21, F23, G12, G14, G15, H11, H14, H15, H23, I11, I14, I15, J11, J14, J15, J21, K11, K14**, and **K15** are marked faulty due to such incorrect specifications of expected intervals. Second, the specified expected interval does not completely lie within the computed bounding interval. As this particular spreadsheet is believed to be fault-free, again the problem lies in the specification of the expected interval. In this case it may be difficult to expect the user to have such a sharp estimation of the expected intervals which lie completely within the bounding intervals. As mentioned in chapter 6 section 6.5.3, if the intersection between the expected interval and the bounding interval is empty or the width of the intersection is less than the other part of the expected interval, then it is appropriate to generate symptoms of faults. On the other hand, if the width of the intersection is larger than the other part of the expected interval, then it is legitimate to make further investigations (for example, compare the expected and bounding interval values of cell I4).

	A	B	C	D	E	F	G	H	I	J	K
1											
2	**Rentals**							1997	1998		
3	Status: May 99					Operating Cost/year		91194.2	95331.88		
4						Operating Cost/month		7599.51667	7944.32333		
5						OpCost/Mon*m2 '		8.70755275	9.10263344		
6									**OpCost-Rate**		**9.00**
7	Location	Used by	Category	Space. m2	Rent / m2	basic Rent	OperatingCost	intermSum	Charges	VAT	Total
8	1,1a,2	K.B.	C	74.00	17.20	1272.80	666.00	1938.80	1939.00	193.90	2132.90
9	4 + 5	H.G.	C	93.95	17.20	1615.94	845.55	2461.49	2461.50	246.15	2707.65
10	6	A.D.	RW	31.90	54.70	1744.93	287.10	2032.03	2032.11	203.21	2235.32
11	7	V.B.	D	33.80	8.60	290.68	304.20	594.88	594.88	59.49	654.37
12	9a	S.A.	RW	87.45	54.70	4783.52	787.05	5570.57	5570.00	557.00	6127.00
13	13	I.M	A	60.00	34.50	2070.00	540.00	2610.00	2610.00	261.00	2871.00
14	14	A.M	A/3	144.70	11.50	1664.05	1302.30	2966.35	2966.36	296.64	3263.00
15		**regulated space**		**525.80**		**13441.92**	**4732.20**	**18174.12**	**18173.85**	**1817.39**	**19991.24**
16											
17											
18	Box8	M.M	Ind	117.00		4947.00	1053.00	6000.00	6000.00	600.00	6600.00
19	3	P.B.	Ind	34.00	206.60	452.00	306.00	758.00	758.00	75.80	833.80
20		BX	unreg	152.17			1369.53	17700.00	17700.00	3540.00	21240.00
21		**unregul.space**		**303.17**		**5399.00**	**2728.53**	**24458.00**	**24458.00**	**4215.80**	**28673.80**
22		unused		43.78			394.02		0.00		
23		**Total**		**872.75**		**18840.92**	**7854.75**	**42632.12**	**42631.85**	**6033.19**	**48665.04**

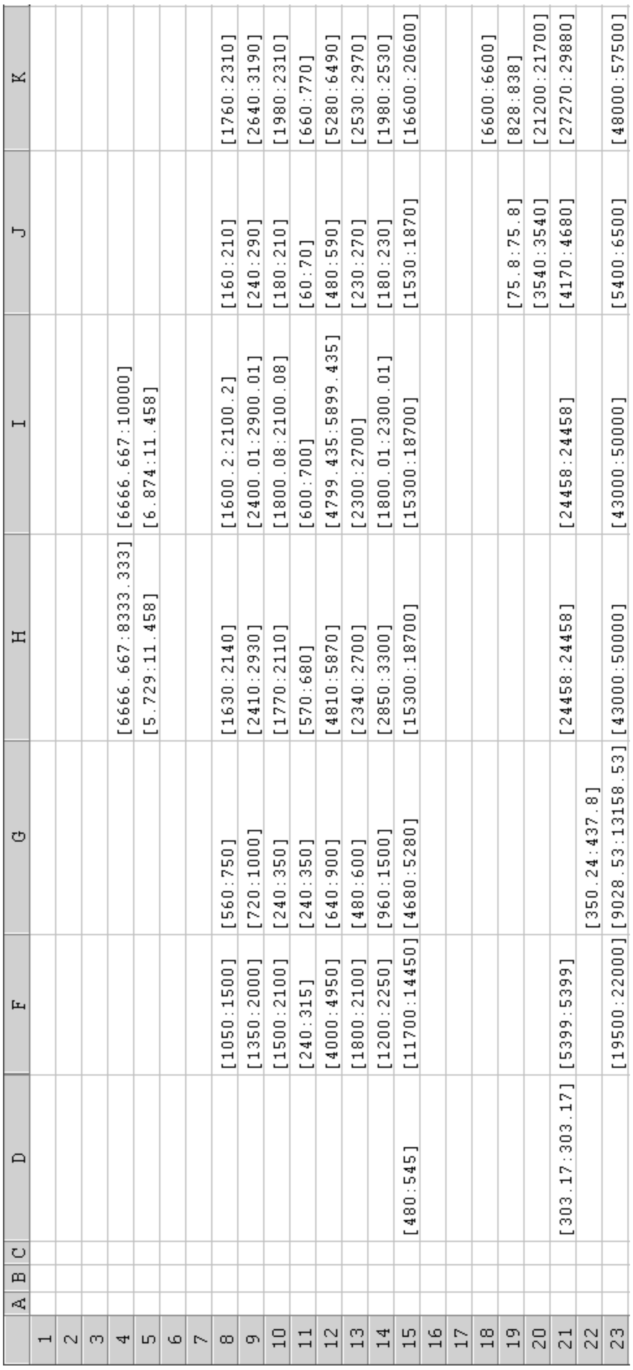
Figure 6.14: A spreadsheet to calculate house rent income

	A	B	C	D	E	F	G	H	I	J	K
1											
2	**Rentals**							1997	1998		
3	Status: May 99					Operating Cost/year		91194.2	95331.88		
4						Operating Cost/month		=H3/12	=I3/12		
5						OpCost/Mon*m2 '		=H4/$D23	=I4/$D23		
6									OpCost-Rate		9
7	Location	Used by	Category	Space. m2	Rent / m2	basic Rent	OperatingCost	intermSum	Charges	VAT	Total
8	1,1a,2	K.B.	C	74	17.2	=D8*E8	=$K$6*D8	=F8+G8	=+H8+0.2	=I8*0.1	=I8+J8
9	4 + 5	H.G.	C	93.95	17.2	=D9*E9	=$K$6*D9	=F9+G9	=+H9+0.01	=I9*0.1	=I9+J9
10	6	A.D.	RW	31.9	54.7	=D10*E10	=$K$6*D10	=F10+G10	=+H10+0.08	=I10*0.1	=I10+J10
11	7	V.B.	D	33.8	8.6	=D11*E11	=$K$6*D11	=F11+G11	=+H11	=I11*0.1	=I11+J11
12	9a	S.A.	RW	87.45	54.7	=D12*E12	=$K$6*D12	=F12+G12	=+H12-0.565	=I12*0.1	=I12+J12
13	13	I.M	A	60	34.5	=D13*E13	=$K$6*D13	=F13+G13	=+H13	=I13*0.1	=I13+J13
14	14	A.M	A/3	144.7	=34.5/3	=D14*E14	=$K$6*D14	=F14+G14	=+H14+0.01	=I14*0.1	=I14+J14
15		**regulated space**		**=SUM(D8:D14)**		**=SUM(F8:F14)**	**=SUM(G8:G14)**	**=SUM(H8:H14)**	**=SUM(I8:I14)**	**=SUM(J8:J14)**	**=SUM(K8:K14)**
16											
17											
18	Box8	M.M	Ind	117		=H18-G18	=$K$6*D18	6000	=+H18	=I18*0.1	=I18+J18
19	3	P.B.	Ind	34	206.6	=H19-G19	=$K$6*D19	758	=+H19	=I19*0.1	=I19+J19
20		BX	unreg	152.17			=$K$6*D20	17700	=+H20	=I20*0.2	=I20+J20
21		**unregul.space**		**=SUM(D18:D20)**		**=SUM(F18:F20)**	**=SUM(G18:G20)**	**=SUM(H18:H20)**	**=SUM(I18:I20)**	**=SUM(J18:J20)**	**=SUM(K18:K20)**
22		unused		43.78			=$K$6*D22		=+H22		
23		**Total**		**=D15+D21+D22**		**=F15+F21+F22**	**=G15+G21+G22**	**=H15+H21+H22**	**=I15+I21+I22**	**=J15+J21+J22**	**=K15+K21+K22**

Figure 6.15: Formula view of rent income spreadsheet

	A	B	C	D	E	F	G	H	I	J	K
1											
2											
3								[80000:100000]	[80000:120000]		
4								[5000:10000]	[6000:10000]		
5								[4.5:10]	[7:10]		
6											[8:10]
7											
8				[70:75]	[15:20]	[1000:1400]	[630:740]	[1600:2100]	[1600:2100]	[160:210]	[1700:2300]
9				[90:100]	[15:20]	[1600:2000]	[810:930]	[2400:2900]	[2400:2900]	[240:290]	[2400:3200]
10				[30:35]	[50:60]	[1500:1800]	[270:310]	[1800:2100]	[1800:2100]	[180:210]	[2000:2300]
11				[30:35]	[8:9]	[300:350]	[270:330]	[600:700]	[600:700]	[60:70]	[700:800]
12				[80:90]	[50:55]	[4000:5000]	[810:870]	[4800:5900]	[4800:5900]	[480:590]	[5300:6500]
13				[60:60]	[30:35]	[1800:2100]	[540:600]	[2300:2700]	[2300:2700]	[230:270]	[2500:3000]
14				[120:150]	[10:15]	[1500:1800]	[1350:1500]	[1800:2300]	[1800:2300]	[180:230]	[2000:2500]
15				[500:700]		[13000:15000]	[6000:10000]	[19000:25000]	[19000:25000]	[1900:2500]	[21000:27500]
16											
17											
18											[6500:7000]
19										[70:80]	[770:880]
20										[3500:4000]	[20000:22000]
21				[300:400]		[6500:7000]		[24000:25000]	[24000:25000]	[3500:4000]	[27000:30000]
22							[300:430]				
23						[19000:22000]	[6000:10000]	[43000:50000]	[44000:50000]	[5500:6500]	[28000:60000]

Figure 6.16: Expected spreadsheet for rent income



	A	B	C	D	F	G	H	I	J	K
1										
2										
3										
4							[6666.667:8333.333]	[6666.667:10000]		
5							[5.729:11.458]	[6.874:11.458]		
6										
7										
8					[1050:1500]	[560:750]	[1630:2140]	[1600.2:2100.2]	[160:210]	[1760:2310]
9					[1350:2000]	[720:1000]	[2410:2930]	[2400.01:2900.01]	[240:290]	[2640:3190]
10					[1500:2100]	[240:350]	[1770:2110]	[1800.08:2100.08]	[180:210]	[1980:2310]
11					[240:315]	[240:350]	[570:680]	[600:700]	[60:70]	[660:770]
12					[4000:4950]	[640:900]	[4810:5870]	[4799.435:5899.435]	[480:590]	[5280:6490]
13					[1800:2100]	[480:600]	[2340:2700]	[2300:2700]	[230:270]	[2530:2970]
14					[1200:2250]	[960:1500]	[2850:3300]	[1800.01:2300.01]	[180:230]	[1980:2530]
15				[480:545]	[11700:14450]	[4680:5280]	[15300:18700]	[15300:18700]	[1530:1870]	[16600:20600]
16										
17										
18										[6600:6600]
19									[75.8:75.8]	[828:838]
20									[3540:3540]	[21200:21700]
21				[303.17:303.17]	[5399:5399]		[24458:24458]	[24458:24458]	[4170:4680]	[27270:29880]
22						[350.24:437.8]				
23					[19500:22000]	[9028.53:13158.53]	[43000:50000]	[43000:50000]	[5400:6500]	[48000:57500]

Figure 6.17: Bounding spreadsheet for rent income

	A	B	C	D	E	F	G	H	I	J	K
1											
2	**Rentals**							1997	1998		
3	Status: May 99					Operating Cost/year		91194.2	95331.88		
4						Operating Cost/month		7599.51667	7944.32333		
5						OpCost/Mon*m2 '		8.70755275	9.10263344		
6									OpCost-Rate		9.00
7	Location	Used by	Category	Space. m2	Rent / m2	basic Rent	OperatingCost	intermSum	Charges	VAT	Total
8	1,1a,2	K.B.	C	74.00	17.20	1272.80	666.00	1938.80	1939.00	193.90	2132.90
9	4 + 5	H.G.	C	93.95	17.20	1615.94	845.55	2461.49	2461.50	246.15	2707.65
10	6	A.D.	RW	31.90	54.70	1744.93	287.10	2032.03	2032.11	203.21	2235.32
11	7	V.B.	D	33.80	8.60	290.68	304.20	594.88	594.88	59.49	654.37
12	9a	S.A.	RW	87.45	54.70	4783.52	787.05	5570.57	5570.00	557.00	6127.00
13	13	I.M	A	60.00	34.50	2070.00	540.00	2610.00	2610.00	261.00	2871.00
14	14	A.M	A/3	144.70	11.50	1664.05	1302.30	2966.35	2966.36	296.64	3263.00
15		**regulated space**		**525.80**		**13441.92**	**4732.20**	**18174.12**	**18173.85**	**1817.39**	**19991.24**
16											
17											
18	Box8	M.M	Ind	117.00		4947.00	1053.00	6000.00	6000.00	600.00	6600.00
19	3	P.B.	Ind	34.00	206.60	452.00	306.00	758.00	758.00	75.80	833.80
20		BX	unreg	152.17			1369.53	17700.00	17700.00	3540.00	21240.00
21		**unregul.space**		**303.17**		**5399.00**	**2728.53**	**24458.00**	**24458.00**	**4215.80**	**28673.80**
22		unused		43.78			394.02		0.00		
23		**Total**		**872.75**		**18840.92**	**7854.75**	**42632.12**	**42631.85**	**6033.19**	**48665.04**
24											
25											

Figure 6.18: Rent income spreadsheet with symptoms of faults

Table 6.4 shows the changes in the number of cells which have symptoms of faults when the comparison criterion varies. The expectation is 100% met means that the expected interval completely lies within the computed bounding interval. 75% of the expectation or more is met means that $w(E \cap B) \geq \frac{3}{4} * w(E)$. Similarly, 50% of the expectation or more is met means that $w(E \cap B) \geq \frac{1}{2} * w(E)$. In table 6.4 the number of faulty cells for each of the criteria is computed by subtracting those faulty cells which are listed above (i.e., 20 faulty cells). Those faulty cells are marked due to the discrepancy between the discrete values and the corresponding expected intervals.

	100%	$\geq$ **75%**	$\geq$ **50%**
Rent Income Spreadsheet	26	10	7
Interest Calculation Spreadsheet(figure 6.13)	3	3	3

Table 6.4: Comparing number of faults detected by varying comparison criterion

Generally, it is expected that as the comparison criterion gets weaker, the number of cells with symptoms of faults decreases. From Rent Income Spreadsheet, we can see that as the comparison criterion gets weaker, the number of false symptoms of faults is reduced. However, for the faulty spreadsheet, Interest Calculation Spreadsheet, there is no change of symptoms of faults.

Returning to the example, the other change that we can observe is when the number of cells which have intervals attached in a formula changes. When many or all of the cells referenced in a formula have intervals attached, then the computed bounding interval will be wider than the bounding interval when less number of cells have intervals attached. This has an effect on the result of the comparison between the expected and the bounding intervals. For example, for the formula $G11 = \$K\$6 * D11$, the bounding interval in figure 6.19 is [270.4, 338] which is narrower than the bounding interval in figure 6.18 which is [240, 350] for the same formula. As a result this formula cell is reported to be faulty in figure 6.19 while it is fault-free in figure 6.18. The formula cell G11 has the same formula and hence the same expected interval in both cases which is [270, 330]. Actually, both spreadsheets are the same except that the input cells in the range **D8..E14** have no intervals attached in figure 6.19 while these same range of cells have intervals attached in figure 6.18.

On the other hand, if all the cells referenced in a formula do not have intervals attached, then the resulting bounding interval will be of width zero and hence has no effect on the comparison between E and B. For example, the formula cells **D15, F8**, and **F12** are marked as non-faulty in figure 6.19. This is because these formula cells have bounding intervals which are of width zero and hence there is no comparison between E and B. However, these formula cells are marked faulty in figure 6.18 since the expected intervals do not lie within the corresponding bounding intervals.

	A	B	C	D	E	F	G	H	I	J	K
1											
2	**Rentals**							1997	1998		
3	Status: May 99					Operating Cost/year		91194.2	95331.88		
4						Operating Cost/month		7599.51667	7944.32333		
5						OpCost/Mon*m2 '		8.70755275	9.10263344		
6									**OpCost-Rate**		**9.00**
7	Location	Used by	Category	Space. m2	Rent / m2	basic Rent	OperatingCost	intermSum	Charges	VAT	Total
8	1,1a,2	K.B.	C	74.00	17.20	1272.80	666.00	1938.80	1939.00	193.90	2132.90
9	4 + 5	H.G.	C	93.95	17.20	1615.94	845.55	2461.49	2461.50	246.15	2707.65
10	6	A.D.	RW	31.90	54.70	1744.93	287.10	2032.03	2032.11	203.21	2235.32
11	7	V.B.	D	33.80	8.60	290.68	304.20	594.88	594.88	59.49	654.37
12	9a	S.A.	RW	87.45	54.70	4783.52	787.05	5570.57	5570.00	557.00	6127.00
13	13	I.M	A	60.00	34.50	2070.00	540.00	2610.00	2610.00	261.00	2871.00
14	14	A.M	A/3	144.70	11.50	1664.05	1302.30	2966.35	2966.36	296.64	3263.00
15		**regulated space**		**525.80**		**13441.92**	**4732.20**	**18174.12**	**18173.85**	**1817.39**	**19991.24**
16											
17											
18	Box8	M.M	Ind	117.00		4947.00	1053.00	6000.00	6000.00	600.00	6600.00
19	3	P.B.	Ind	34.00	206.60	452.00	306.00	758.00	758.00	75.80	833.80
20		BX	unreg	152.17			1369.53	17700.00	17700.00	3540.00	21240.00
21		**unregul.space**		**303.17**		**5399.00**	**2728.53**	**24458.00**	**24458.00**	**4215.80**	**28673.80**
22		unused		43.78			394.02		0.00		
23		**Total**		**872.75**		**18840.92**	**7854.75**	**42632.12**	**42631.85**	**6033.19**	**48665.04**
24											

Figure 6.19: A spreadsheet with symptoms of faults when no intervals are attached to input cells

6.8 Summary

Software testing is a complex and time consuming activity. To aid testers, a variety of techniques has been proposed and is being used at various stages of development. The demand for such techniques may be even higher for end-users who are not trained in the formal process of software development. In this chapter, we have presented an approach to provide a testing methodology for spreadsheet users. This approach incorporates a conventional software testing technique (i.e., symbolic testing) and interval analysis and is proposed by taking into consideration the expertise of users and the inherent characteristics of spreadsheets. As such it is a user-centered approach.

Unlike symbolic testing, which requires expressing formulas only in terms of input variables, interval-based testing uses intermediate variables for the purpose of narrowing down the computed interval. In addition, while symbolic testing is used to validate a formula for any possible values of the input variables, interval-based testing requires the values of the variables to be expressed as intervals and validity is determined based on the intervals provided.

Interval-based testing focuses on the functionality of spreadsheet formulas instead of the internal structure of a spreadsheet program (i.e., it is not based on code coverage). The observation that spreadsheets are mainly used for numerical computations enables us to introduce the idea of interval analysis to spreadsheet testing. It requires the user to specify input and expected intervals for desired input and formula cells respectively. This will be documented in a behind-the-scene spreadsheet and used to perform interval computations during the verification of a given spreadsheet. In addition, the expected intervals provided by the user are verified for reasonableness using interval analysis. Furthermore, the interval-definition phase also serves as a kind of manual review process since it requires the user to check and think about the functionality of the particular formula. Despite the requirement to attach intervals, the proposed approach based on interval analysis does not require any knowledge of software testing.

As observed from different examples, it seems reasonable to make further investigation in order to determine the existence of symptoms of faults when the expected interval and the corresponding bounding interval agree to a large extent. There are different factors which affect total satisfaction of user expectation. First, some of the cells involved in the computation of a formula may not have intervals attached resulting in a relatively narrow bounding interval. Second, it may be difficult for a user to specify expected intervals that completely lie within the computed bounding intervals. Therefore, if the percentage of the users expectation met is reduced from 100% (an ideal value) to some reasonable value (say 75%), then some false symptoms of faults may be avoided.

Chapter: 7
Fault Tracing

After testing is performed, the next issue is how to locate the actual faults in a program. This chapter presents a technique for tracing faults in a spreadsheet program. Section 7.1 discusses the general problem of fault tracing in relation to conventional software debugging techniques. Section 7.2 discusses the fault tracing strategy to be used in spreadsheets. While tracing for faults, there is a need for minimizing the search region. This requires the use and computation of priority values which is presented in section 7.3. A simple example and fault tracing algorithm are described in this same section. The main points of the fault tracing approach are summarized in section 7.4.

7.1 Fault Tracing Background

Once the existence of symptoms of faults is detected, the next task is to find the location of the actual faults. A testing system can not exactly indicate the location of faults, it rather provides a hint or symptom of a fault. However, a testing system can facilitate the search for the location of faults by providing testing information about the possible paths that lead to the likely fault location. Fault tracing is the process of identifying the location of faults in a program. Generally, fault tracing in conventional software debugging involves program slicing techniques to minimize the search for the potential faults [1, 2, 29, 55]. The first important information needed in fault tracing is static backward slice which contains all variables which may affect the variable at which a symptom of fault is detected. Since the symptom of fault is generated based on a particular test case, those variables which are directly and indirectly involved in the current computation will contain the faulty variable provided that the fault is not due to missing statements. This requires the computation of dynamic backward slices. In order to reach the potential faulty variables, further reduction of the dynamic backward slice should be made using the technique of dicing. Dicing is carried out by removing the sub-slice corresponding to correct variables. However, the use of dicing imposes some preconditions to be satisfied in order for the resulting dice to contain the fault [61]. For example, dicing assumes that only one faulty variable exists in the dynamic backward slice. In addition, it assumes that if a variable is faulty then all variables in the dynamic forward slice of that variable are faulty. This avoids the situation where faults compensate in between. In other words, dependents of a faulty variable should not provide a correct result by compensating through another fault. Hence, the general process of fault tracing in conventional software can be described as follows.

Static slice $\longrightarrow$ Dynamic slice $\longrightarrow$ Dice $\longrightarrow$ Potential location of fault

A similar procedure can be used for fault tracing in spreadsheets. However, in our approach we do not impose the requirement that the dynamic backward slice contains only one faulty cell. There can be several cells in the dynamic backward slice of a faulty cell which are marked as faulty. Cell marking is performed based on the interval-based testing methodology discussed in chapter 6. The fault tracing procedure uses the backward slice and the cell marks recorded by the interval-based testing system.

A fault localization technique proposed by DeMillo et al. [29] was based on the analysis of the steps used by experienced programmers in debugging. Following a similar procedure, the spreadsheet fault tracing process contains the following steps:

1. Determine the cells involved (directly and indirectly) in the computation of an incorrect formula (i.e., look backward)

2. Select suspicious cells

3. Make hypotheses about suspicious cells

Step 1 above requires the computation of backward slices with respect to a faulty cell for which we want to locate the source of the fault. Step 2 requires the identification of those cells which have a likelihood of propagating faults through the data flow. These cells are marked by the comparator during the verification process. Step 3 requires reasoning about the most influential cell. In other words, this step involves the computation of the priority values of those suspicious cells and the identification of the one which has the highest likelihood of contribution to the faulty cells in the dynamic backward slice.

Therefore, the main problem in fault tracing in spreadsheets is the identification of the most influential faulty cell(s) when the fault is propagated. If the fault is local, meaning that either the formula or the expected interval is specified incorrectly, then the fault can be fixed by examining the faulty cell itself. The assumption in this approach is that if the most influential faulty cell is found, then correcting this cell may correct many of the dependent faulty cells in the data dependency graph thereby reducing the debugging process. To address this problem, we propose an approach using priority setting based on the number of incorrect precedents and dependents. To do so, we rely on the dynamic backward slice of a given faulty cell as we are dealing with propagated faults. Therefore, we make the assumption that the fault is not due to omission of cell(s) in the formula. If the fault is due to omission of a cell, then we can not trace the fault in the backward slice with respect to the faulty formula cell under consideration.

A similar approach was proposed by Reichwein et al. [96] for the purpose of debugging Form/3 programs. In that approach, a user marks cells as correct and incorrect and based on the all-uses dataflow test adequacy criterion, the degree of testedness of formulas is computed. Cells are given different colors based on their degree of testedness. Fault tracing is performed based on the degree of testedness of cells and by computing fault likelihood of cells. Fault likelihood is computed based on the number of correct and incorrect dependents of a cell. During the fault localization process, for the cell under consideration, those cells in the dynamic backward slice will be highlighted in different colors based on their degree of testedness. Further process of fault localization is carried out by performing testing using additional test cases.

In our approach, a priority value is computed based on the verification status of precedent cells and in some cases based on dependent cells. Let C be a given cell. Precedent cells of C are those cells which are directly (direct precedents) or indirectly (simply precedents) referenced in the formula of C. Similarly, dependents of C are those cells which reference C directly (direct dependents) or indirectly (simply dependents) in their formula.

7.2 Fault Tracing Strategy

The fault tracing strategy for spreadsheets uses information from different sources to locate the most influential faulty cell. The first information that we need is the dataflow information. This information is already available since it is used by the spreadsheet language during the evaluation of formulas. For example, in Microsoft Excel, this information is used by the auditing tool to show the backward and forward slices of cells of interest. While traversing the backward slice, we need a mechanism of selecting the cells which have a likelihood of being the most influential faulty cell. This information can be obtained from the testing system as cells are marked with different colors depending on the existence of symptoms of faults. Hence, cell marks are used to guide the search process. Based on the type of cell marks encountered, during traversing the backward slice, priority values are computed to guide the search to the path where the most influential faulty cell may be located. Therefore, the fault tracing strategy uses dataflow, cell marks, and priority values to locate the most influential faulty cell.

7.3 Computation of Priority Value

Priority values are used to indicate the path through which the search should focus. A definition of priority value is given below.

Definition: (Priority Value - PV)
Priority value of a cell is a value assigned to the cell based on its verification status. A higher priority value indicates that the cell has a higher likelihood of containing the most influential faulty cell.

During the verification process, cells in a spreadsheet are categorized into three groups. These are cells with symptoms of faults, cells without symptoms of faults, and cells which are unchecked. Let C represent a cell without a symptom of fault, E represent a cell with a symptom of fault, and U represent a cell which is unchecked. Using the verification information of cells, the priority values are related as follows.

$$\boxed{PV(C) \ < \ PV(U) \ < \ PV(E)}$$

The assignment of priority is also based on the contribution of a cell to incorrect dependents. In other words, a faulty cell which has more incorrect dependents is more influential than the one with few incorrect dependents. Furthermore, those faulty cells which are at a higher level of the data dependency graph (i.e., near the input cells) are more influential than those at the lower level of the data dependency graph. Therefore, correcting those faulty cells at the highest level of the data dependency graph may correct those cells showing incorrect values but being not actually faulty which are dependent on the corrected cell thereby reducing the effort of the debugging process. Under some circumstances, for a faulty cell under consideration, there may be more than one influential cell which have the same priority value. In this case, all those cells which have the same priority value are chosen.

7.3.1 Example

Let us once again consider the interest calculation spreadsheet which was presented in chapter 6 section 6.5.1. For the sake of easy reference, the spreadsheet is presented again in figure 7.1.

	A	B	C
1	deposit	rate	payment
2	3000	0.03	=A2*B2
3	5000	0.035	=A3*B3
4	8000	0.05	=A4*B3
5	10000	0.06	=A5*B4
6			=SUM(C2:C5)
7			

(a)

	A	B	C
1	deposit	rate	payment
2	3000	0.03	90
3	5000	0.035	175
4	8000	0.05	280
5	10000	0.06	500
6			1045
7			

(b)

Figure 7.1: Interest calculation spreadsheet with symptoms of faults

Suppose for the spreadsheet in figure 7.1, the user wants to trace the most influential faulty cell for the final result of the computation in cell C6. Actually, the first thing to examine is the faulty cell itself. If the formula and the expected interval attached are correct, then the fault is due to referencing a faulty cell. In such cases, we need to trace the source of the fault. Returning to the example, the direct precedents of cell C6 are cells C2, C3, C4, and C5 (see figure 7.2). The direct precedents fall into the two categories correct (those without symptoms of faults) and faulty. The faulty category which contains cells C4 and C5 is the candidate for further investigation. The next task is to identify which one of C4 and C5 contains the most influential faulty cell. Since they have equal priority values, then we check the number of their faulty direct precedents and dependents. Each of them have no faulty direct precedents, but one faulty direct dependent. Further check up reveals no difference in the priority values as the root of the data dependency graph is reached which contains input cells.

Therefore, both cells C4 and C5 are the most influential faulty cells and as a result, whenever the user requests for the most influential faulty cell for cell C6, the system highlights the cells C4 and C5. However, if C4 and C5 had other faulty direct dependents, then we need to consider also the number of their direct dependents.

Suppose a cell K depends on C5 and has a symptom of fault. As we have seen before, the number of faulty direct precedents of C4 = number of faulty direct precedents of C5 = 0. Now, by considering the number of faulty direct dependents of each cell, we get $NFD(C4) = 1 < NFD(C5) = 2$ which indicates that C5 should be chosen as the most influential faulty cell. Note that NFD stands for number of faulty direct dependents.

While using the number of faulty direct dependents, there are two possibilities to consider.

- using the the number of faulty direct dependents combined with the number of faulty direct precedents
- using the the number of faulty direct dependents when the number of faulty direct precedents are equal

If we use the first choice, we may find the most influential cell without going far in the data dependency graph. This influential cell may not be the most influential for the faulty cell under consideration but contributes to many other faulty dependent cells. Therefore, correcting this cell may also correct many other cells which are not in the dynamic backward slice of the cell under consideration. This option identifies the most influential cell in terms of the number of incorrect dependents of a cell.

If we use the second choice, then we can reach to the most influential cell with respect to the cell under consideration which is at a higher level in the data dependency graph. Therefore, correcting this cell may correct many cells in the dynamic backward slice of the cell under consideration. Though both options provide the possibility of correcting many cells, we prefer to locate the most influential cell using the second option as this identifies the most influential cell for the cells in the dynamic backward slice. In case two or more faulty cells have equal priority after adding the number of their faulty direct dependents, then one of them will be chosen arbitrarily provided that they have faulty precedents. On the other hand, if they do not have faulty precedents, then all of them are selected as the most influential faulty cells.

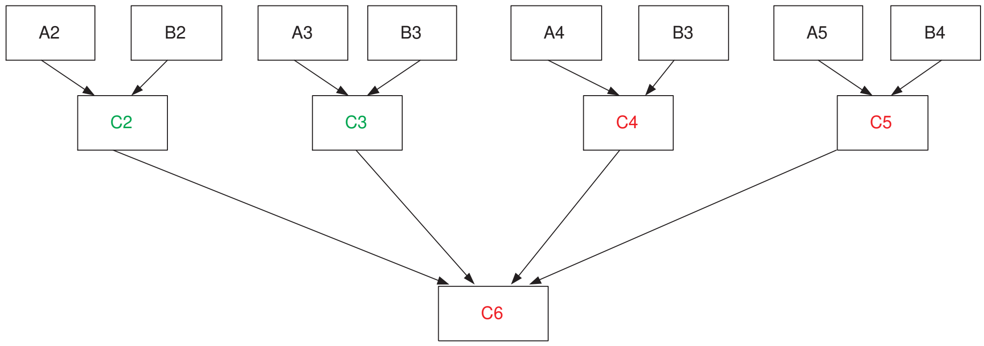


Figure 7.2: Data dependency graph for interest calculation spreadsheet

Suppose the spreadsheet in figure 7.1(a) is modified as shown in figure 7.3.

	A	B	C
1	deposit	rate	payment
2	3000	0.03	=A2*B2
3	5000	0.035	=A3*B3
4	8000	0.05	=A4*B3
5	10000	0.06	=A5*B4+C4
6			=SUM(C2:C5)
7			

Figure 7.3: Modified interest calculation spreadsheet

The corresponding data dependency graph of the modified spreadsheet is shown in figure 7.4. In this modified version, C5 references C4 thereby introducing a new data dependency relationship. Similar to the previous case, the faulty category which contains cells C4 and C5 is the candidate for further investigation. C4 and C5 are both faulty. The next task is to compare the number of their faulty direct precedents. The number of faulty direct precedents of C4 is 0 since the two precedent input cells are correct. On the other hand, the number of faulty direct precedents of C5 is 1 since it contains C4 as a precedent. Therefore, the path to C5 should be followed to locate the most influential faulty cell. Again, we need to compare the priority values of the precedents of C5. C4 turns out to have a higher priority value and hence the path to C4 should be followed. Comparing the precedents of C4, we find no faulty precedents. Therefore, C4 is selected as the most influential faulty cell.

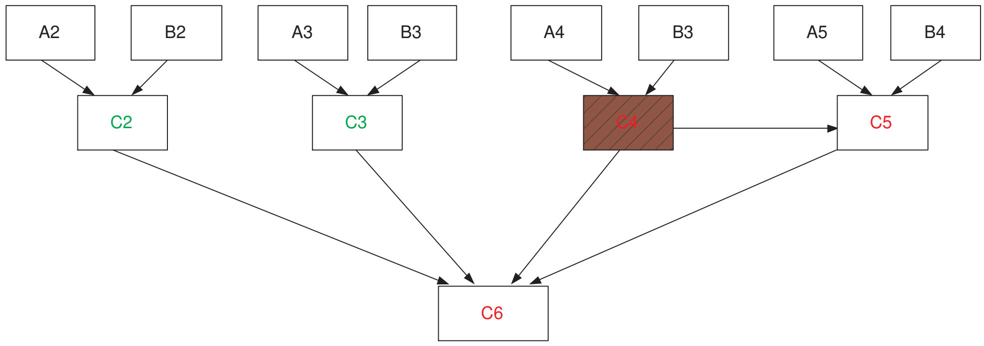


Figure 7.4: Modified data dependency graph for interest calculation spreadsheet

7.3.2 Fault Tracing Algorithm

The algorithm for identifying the most influential faulty cell is presented in algorithm 6 for propagated faults. Let C be the faulty cell for which we are interested in to identify the most influential faulty cell(s).

Algorithm 6 Algorithm to identify the most influential faulty cell

1: Get precedents of C.

2: Group precedents of C according to their priority values.

3: Let G be the group that contains faulty cells such that $G = \{C_1, C_2, ..., C_n\}$. If $G = \emptyset$, then C is the most influential cell.

4: Compute the number of faulty direct precedents of each of the precedents of elements of G. For those with equal number of faulty direct precedents, compute the number of their faulty direct dependents.

5: Repeat step 4 until the cell with maximum number of faulty precedents (and dependents) is found.

6: Choose the path to the cell with maximum number of faulty (direct) precedents (and dependents). Let C be the cell with maximum number of faulty (direct) precedents (and dependents).

7: Repeat steps 1 to 6 until no faulty direct precedents are found or the most influential cell(s) is found.

7.4 Summary

After testing is performed, the task of locating the actual faults is handed over to debugging. Debugging involves the identification of the location of faults and fixing the fault. In spreadsheet programs, the identification of cells with symptoms of faults and those without symptoms of faults is carried out by the testing system (i.e., interval-based testing). To fix the faults, we need to examine each cell which has a symptom of fault. This is inefficient. So, examining and fixing the faulty cells should be done systematically. In other words, as spreadsheet programs are dataflow-driven, faults are propagated in the direction of dataflow. Therefore, we need a mechanism of identifying the most influential faulty cell in the data dependency graph so that correcting it may correct many cells in the data dependency graph thereby simplifying the debugging process.

In this chapter, we have presented a technique for the identification of the most influential faulty cell for a given faulty cell which has a propagated fault. Unlike conventional software fault localization techniques which apply dicing, we do not limit the number of faulty cells in the dynamic backward slice of the cell under consideration to one. Several cells with symptoms of faults can appear in the dynamic backward slice of a given faulty cell. For the identification of the most influential faulty cells, the fault tracing strategy uses the dataflow information which is available from the spreadsheet language, the cell marks obtained from the testing system, and the priority values of cells. Path selection is based on the computation of priority values of precedent cells. Whenever precedent cells have equal priority values, the priority values of their direct dependents are also taken into consideration.

Chapter: 8

Implementation

In order to demonstrate the effectiveness of the interval-based testing methodology, a prototype tool is developed. The tool incorporates a parser, an interval arithmetic module, a comparator, and a fault tracer. This chapter describes the design and implementation of the prototype for the interval-based testing methodology. Section 8.1 describes the environment under which the prototype is implemented and section 8.2 presents the architecture defined to integrate the interval-based testing methodology on top of the Excel spreadsheet system. Also, a description of the components of the architecture is given in this section. A summary of the main points of the implementation is given in section 8.3.

8.1 Description of Environment

The pilot environment chosen for the demonstration of interval-based testing is MS Excel under the Windows environment. We use Visual Basic which is the programming language provided to enhance Microsoft Office applications. In order to integrate interval-based testing on top of the MS Excel environment, we make use of the object model provided by the system. MS Excel has got a variety of predefined objects such as Workbook, Worksheet, and Range(cell).

8.2 Architecture

To integrate the interval-based testing methodology on top of spreadsheet systems, an architecture is defined. Figure 8.1 depicts the general architecture to integrate the interval-based testing methodology on top of the MS Excel environment. This architecture indicates the dataflow between the different components of the system.

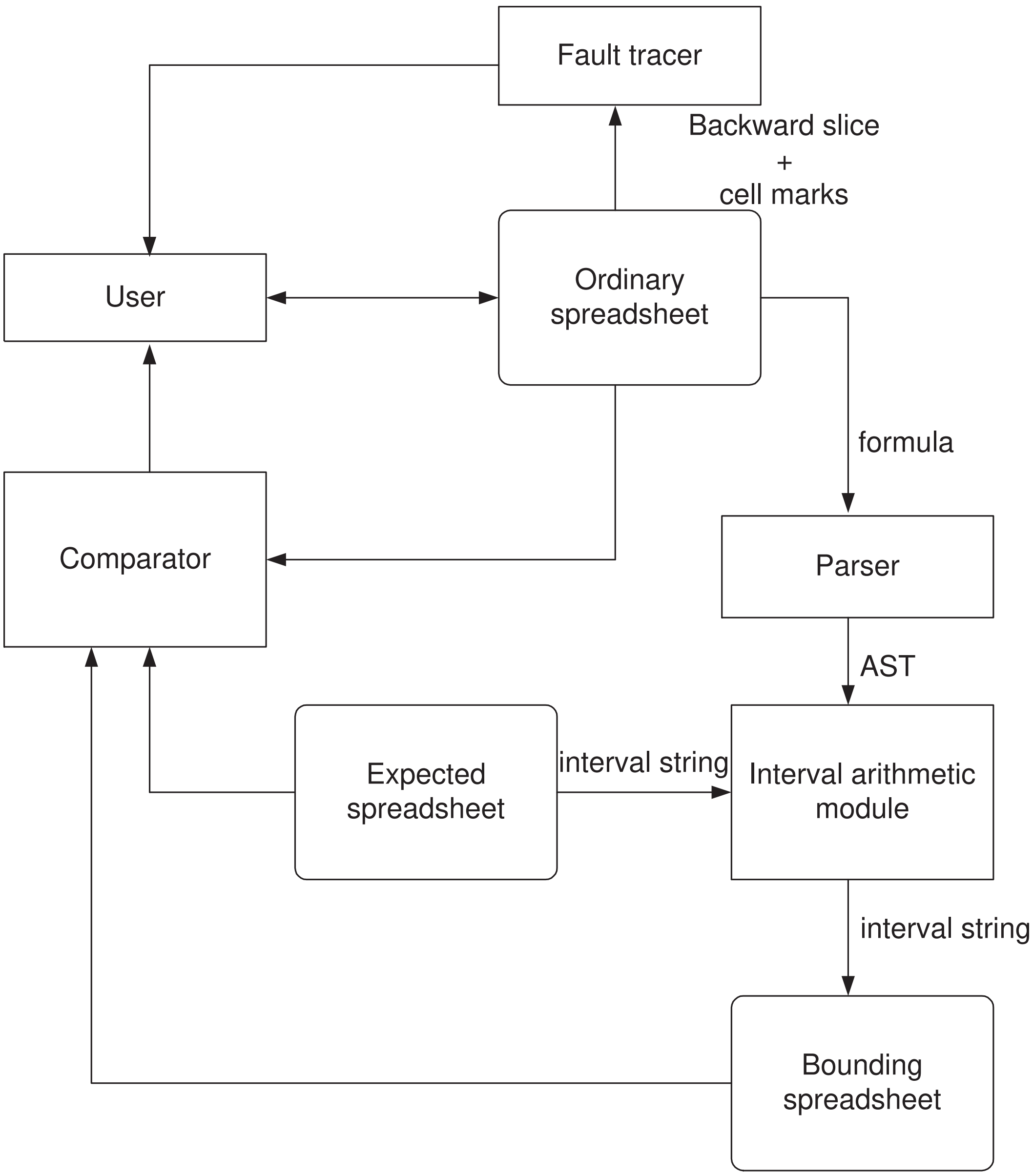


Figure 8.1: Architecture for the interval-based testing methodology

As can be seen from figure 8.1, the user interacts only with the ordinary spreadsheet. In other words, development of a spreadsheet program is carried out in the usual way. However, when the user wants to attach an interval to a given cell, the user selects the cell and chooses a menu command from the **Interval** menu through which interval-based testing features are available. The attached interval is then stored in the expected spreadsheet as a string with the same coordinate as the corresponding cell selected in the ordinary spreadsheet (see figure 8.2). This can be done also for a group of cells when they are intended to have the same interval.

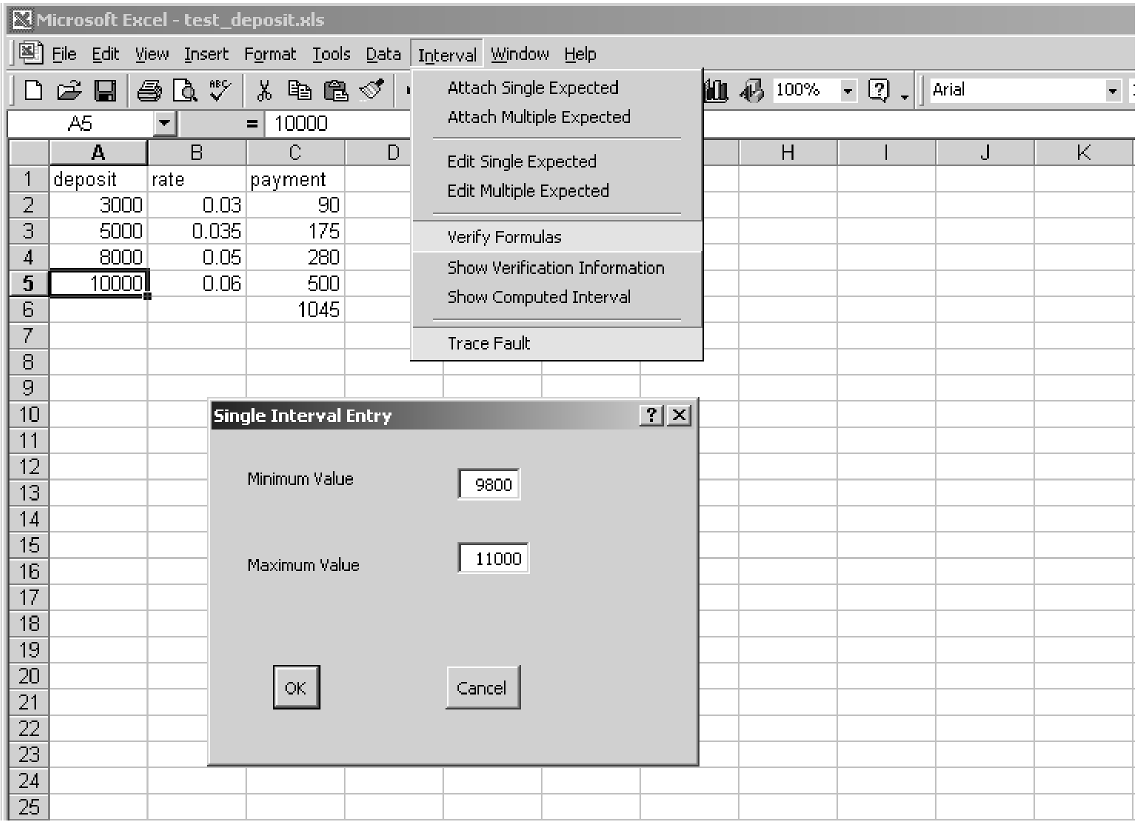


Figure 8.2: Attaching an interval to a numeric cell

Whenever a user is interested to verify a particular formula cell or group of formula cells, the desired cells are selected and a **Verify Formulas** command issued. Subsequently, the testing system parses each formula and generates the abstract syntax tree. While traversing the abstract syntax tree, the infix form of the arithmetic expressions is converted into postfix form to facilitate the computation. During interval computation, the formulas are evaluated based on interval arithmetic semantic of the operators used. Finally, the resulting value of the interval computation which is the bounding interval for the respective cell is stored in the bounding spreadsheet as a string. Once the necessary values are available from the three sources, namely, spreadsheet computation, user expectation, and interval computation, the comparator determines the existence of symptoms of faults.

8.2.1 Parser

The parser used in this tool is a subset of a parser that could be used for the spreadsheet language. Microsoft Excel allows to perform a variety of complex operations, but our parser is not designed to handle all such operations. For the demonstration of our approach, we selected basic operations which can be used in an ordinary spreadsheet development. Actually, it would have been appropriate to use the same parser used by the spreadsheet langauge. However, such parsers are not publicly available as they are used in commercial products.

8.2.2 Interval Arithmetic Module

The interval arithmetic module receives the abstract syntax tree from the parser and interval values from the expected spreadsheet. Based on the abstract syntax tree of a given formula, values for the cell addresses involved in the formula are taken from the expected spreadsheet. After evaluating the formula, the resulting interval is written in a cell in the bounding spreadsheet whose coordinate is the same as the coordinate of the cell from which the formula is extracted. Again, the interval arithmetic module is not intended to cover all function evaluations that are provided by MS Excel. This module is limited to the basic arithmetic operators and spreadsheet functions. For example, it does not handle evaluations of trigonometric or logarithmic functions. Similarly, the current implementation supports interval computation only for simple IF functions. Nested IF functions are currently not supported. Though conceptually the same procedure can apply for nested and multiple IFs, the specification of expected intervals for nested and multiple IFs is a tricky task.

8.2.3 Comparator

The comparator is the component of the system which decides whether a given cell has a symptom of fault. This is accomplished by comparing the values from the ordinary spreadsheet, expected spreadsheet, and bounding spreadsheet. The comparator converts the interval strings which are extracted from the expected spreadsheet and bounding spreadsheet into interval data types to perform comparisons. Whenever there is a discrepancy between the values, a cell is marked red indicating the existence of a symptom of fault. Otherwise, the cell is marked yellow indicating the conformance of the values. After the selected region of cells are marked, the next task is how to fix the faults in a shorter time possible. Therefore, we have to find a mechanism that guides the user in locating the most influential faulty cells to be corrected first. This task is handled by the fault tracer.

8.2.4 Fault Tracer

As discussed in chapter 7, the fault tracer is used to locate the most influential faulty cell which has the highest contribution of faultiness in the data dependency graph. The most influential faulty cell is the first candidate to make corrections with the assumption that correcting this cell may correct many of those faulty cells which are dependent on this cell.

The fault tracer, based on the algorithm given in chapter 7, traverses through the dynamic backward slice of the selected cell and identifies the most influential cell which needs to be fixed in order to shorten the debugging process. In MS Excel spreadsheet system, the backward slice of a cell is computed by collecting the precedents. This information is already recorded during the development of a spreadsheet program. In addition to the dynamic backward slice corresponding to the desired cell, the cell marks in the dynamic backward slice are used to search for the most influential cell. After the most influential cell is corrected, testing is performed again to see the effect of the correction. This process is carried out iteratively until all the desired faulty cells are fixed.

8.3 Summary

We have developed a testing system for spreadsheets which is based on symbolic testing and interval analysis and described the architecture of the interval-based testing methodology which enables us to integrate the system on top of spreadsheet systems. The testing system incorporates a parser, an interval arithmetic module, a comparator, and a fault tracer. The parser takes spreadsheet formulas specified by the user as input and generates the corresponding syntax tree. The arithmetic module evaluates the formula using the abstract syntax tree and substituting intervals instead of discrete values. By extracting the values from the three sources, the comparator decides the existence of symptoms of faults in formulas. Finally, based on the dynamic backward slice of a cell and the associated cell marks, the fault tracer identifies the most influential faulty cell.

Though not yet experimentally validated by using user participants who are real application experts, the integration of the tool to the familiar spreadsheet system and its usability without requiring any concept of conventional software testing are important contributions towards achieving the goal.

CHAPTER: 9
Conclusion and Further Work

9.1 Conclusion

The fact that spreadsheet programs are software might give the impression at first sight to apply directly conventional software testing techniques to spreadsheets. However, conventional software testing techniques, by their very nature, are targeted to professional programmers. An analysis on the suitability of conventional software testing techniques for spreadsheets has already shown the inadequacy of these techniques to spreadsheets. The implication obtained out of the analysis is that testing approaches for spreadsheets should take into account the inherent characteristics of spreadsheets as well as the conceptual models of spreadsheet developers. Therefore, interval-based testing is proposed based on the premises that spreadsheets are mainly used for numerical computations and spreadsheet developers are end-users who are not expected to follow the formal process of software development.

Focusing on the numeric properties of spreadsheets, interval-based testing tries to establish a connection between the users numeric expectations relative to individual cells and how these expectations lead to intervals of potential values. Discrepancy between computed intervals and expected intervals will inform the user about divergences between the mental model and the specified model.

Despite the requirement to attach intervals, the proposed approach does not require any knowledge of conventional software testing. As such it is a user-centered approach. It is user-centered also in so far as the specification of expected intervals requires the spreadsheet developer to reconsider the sheet from a perspective different from the coders perspective.

The interval-based testing methodology provides the following features.

- The methodology can be used by end-users without any requirement of software testing background and we believe that this methodology will have a better chance to be practiced by the large community of spreadsheet users.
- The comparison criterion provides a two-level checkup. The comparison between the value computed by a spreadsheet and the expected interval specified by the user is a verification of spreadsheet instances whereas the comparison between the expected interval and the computed interval provides a more general verification at the spreadsheet level.
- Symptoms of faults due to incorrect specification of expected intervals can be used as a reminder to revise the goal and to have a better understanding of the program.
- The expected intervals specified by the user are not taken for granted, rather the reasonableness of the expected intervals is verified using interval analysis.
- The interval-definition phase once again requires the user to check and think about the functionality of the particular formula cell and as a result this phase also serves as a kind of manual review process giving the methodology a combined feature of automatic testing and manual reviews.

9.2 Summary

There are two main aspects that this thesis has dealt with in order to come up with an appropriate testing methodology for spreadsheets. First, although spreadsheet programs are software too, they are different from conventional software. This inspired us to investigate the differences and similarities between spreadsheet and conventional software. Second, spreadsheet developers are mainly end-users who are not expected to know the software engineering theory, particularly testing techniques. Therefore, the challenge has been to come up with a solution which entertains these aspects. Interval-based testing methodology attempted to sufficiently address these issues.

Interval-based testing focuses on the functionality of spreadsheet formulas instead of the internal structure of a spreadsheet program (i.e., it is not based on the code coverage criterion). It requires the user to specify input and expected intervals for desired input and formula cells respectively. This will be documented in a behind-the-scene spreadsheet and used to perform interval computations during the verification of a given spreadsheet. In addition, the expected intervals provided by the user are verified for reasonableness using interval analysis. The approach provided is thus essentially a kind of stratified plausibility check based on the consistency of legitimate boundaries users might specify for computations.

The main contributions of this work are listed below.

- A user-centered approach for the verification of spreadsheet programs is proposed and implemented.
- An architecture for the integration of the methodology on top of the familiar spreadsheet system is defined.
- A fault tracing algorithm is provided. Once verification is carried out, the next task is to find out the location of the most influential faulty cells so that debugging is performed with a least effort. The fault tracing algorithm provided serves this purpose.
- The introduction of interval computations on spreadsheets for the purpose of testing might give other opportunities for the applications of intervals in spreadsheets. This basically has a promising line as intervals are used for problems which require numerical computations and spreadsheets are one such applications dealing with numerical computations.

9.3 Limitations

Interval-based testing is not without limitations. The limitations are lack of verification generalization for structurally same formula cells, extra width in the bounding intervals, variation of bounding intervals for aggregation operations, handling of nested and multiple IFs, and difficulty of generating expected intervals for a formula cell.

- Lack of verification generalization

  In interval-based testing, structurally same formulas that are used in a group of cells require checking separately. That is, though the formulas are structurally the same, the computational values could be different, requiring different expected intervals. Normally, since these formulas have the same structure, it would seem appropriate to verify one formula and propagate verification information to the other formulas in the group. This requires structural consideration instead of the functionality of the cells. While dealing with structural investigations, this problem can be handled if a relationship between the source and copies of a formula is established. A similar approach can also be used by identifying logical areas which contain cells that have the same formula structure as in [5]. The approach used in [13] considers such structural sameness, but does not solve the problem as a whole since it lacks functional consideration. Therefore, an approach that combines structural and functional considerations could be effective in addressing this problem. Based on the functionality of cell formulas, interval-based testing methodology addresses this problem partly by enabling users to specify the same expected interval for a group of formula cells which have the same functionality and whose values are closer to each other in magnitude. These cells are intended to represent similar computations.

- Bounding intervals are usually wide

  As interval arithmetic provides global maximum and minimum of possible arithmetic on intervals, the bounding interval could be very large in some situations. For example, division that involves intervals containing zero value results in infinity end points. Arithmetic operations such as summation of values of cells in a large area of cells may result in a wide bounding interval. This problem will not have a solution as this depends on the definitions of interval arithmetic. Similarly, a dependency problem results in too much width in the computation of the bounding interval. A dependency problem occurs when a single variable occurs more than once in a formula. There are some mathematical methods of solving the dependency problem such as branch-and-bound algorithms [36]. However, the development of algorithms for such tasks is beyond the scope of this work. Hence, in both cases, incorrectly specified expected intervals may lie easily within the computed interval. In such cases, we rely only on the result of the comparison between spreadsheet computation and the expected interval specified by the user.

- Variation of bounding intervals for aggregation operations

  This problem arises as a result of using intermediate cells' interval values during the computation of the bounding interval of a given formula cell. Intermediate cells have been used for the purpose of narrowing down computed bounding intervals. However, if a given formula can be computed using different alternatives such as sum of column sums or sum of row sums, the resulting bounding intervals could be different. In such cases, the comparison between the expected interval associated with the given formula and the bounding intervals (which can be computed using the different alternatives) may give a symptom of fault for one alternative while there is no symptom of fault for the other alternative.

- Nested and multiple IFs

  As described in chapter 6 section 6.4.2, interval analysis for spreadsheets containing nested IFs is not convenient especially from the users perspective. Nested IFs require the specification of expected intervals corresponding to each possible branch that will be executed. This requires an appropriate user interface corresponding to the number of branches. To do so, the formula containing nested IFs should be parsed and the number of branches computed. This is a dynamic activity which varies depending on the type of nested IFs involved in a formula. Multiple IFs also pose a similar problem. The number of expected intervals depends on the number of possible combinations of branches which will be executed. Therefore, the current implementation supports only simple IFs. It does not support nested and multiple IFs.

- Difficulty of specifying expected intervals

  It may be argued that it is difficult to imagine a reasonable expected interval when the formula involves a large number of cell references. For example, the use of aggregation functions over a large area of cells like the sum of 100 items with each cell value large in magnitude. Well, the question to be raised here is that "How do users verify such cells using other methodologies if they do not have some range of values in mind that is acceptable?". Since expected intervals represent the expected behavior of computations, they should be available before computational formulas are coded and executed. Beizer [7] has described this situation as follows: "In real testing the outcome is predicted and documented before the test is run. If a programmer can not reliably predict the outcome of a test before it is run, then that programmer does not understand how the program works or what it is supposed to be doing. The tester who can not make that kind of prediction does not understand the program's functional objectives." In a similar fashion, the developer of a spreadsheet is expected to be able to predict the outcome of a formula as long as (s)he understands the functionality of the spreadsheet.

9.4 Further Work

Future research work can be done in different directions to improve the interval-based testing approach. For example, development of automatic verification, automatic generation of test cases, and carrying out controlled experiments.

- There are two modes of verification: by request and automatic. By request verification is used to check suspected cell or group of cells when the need arises. This mode of verification supports spreadsheet-based verification. This mode of verification is already implemented. The other alternative is to perform verification automatically following ordinary spreadsheet formula evaluation. In attempting to introduce automatic verification, a major problem which will be encountered is to maintain the consistency between the ordinary spreadsheet computation and the interval computation. In ordinary spreadsheet computation, whenever a change is made to a cell value, its dependents are automatically updated. To maintain both computations consistent, a propagation engine must be implemented as a separate module so that whenever changes are made to interval values or to the ordinary spreadsheet, all dependent cells are re-calculated. Following this, the comparator should be run for those dependent cells to update their status of verification. It will be mainly applicable during spreadsheet maintenance and cell-based verification.

Both modes of verification have their own advantages and drawbacks. The advantage of by request verification is that interval computation and verification are performed only when needed thereby reducing the computational cost. The advantage of automatic verification is that it complies with the interactive nature of the spreadsheet systems. Automatic verification follows the "immediate visual feedback" characteristics of spreadsheet systems. The drawback of by request verification is that even though a formula cell is identified erroneous, there is no sign how this cell may affect dependent cells unless those dependent cells are included in the area selected for verification. So, the main drawback associated with this mode of verification is the inability to propagate symptoms of faults. The disadvantage associated with automatic verification is cost of computation of bounding intervals and comparisons for those dependent cells (similar to automatic re-calculation of ordinary spreadsheet) whenever there is a change in one of the precedent cells. Though automatic verification has the advantage of automatically updating the verification status of cells, it might be messy to see a lot of symptoms of faults propagated through the data dependency graph while a single change is made to some numbers. However, the user can be given control of the situation by providing options as to which mode of verification to use.

- Even though interval-based testing is not code coverage-based, automatic generation of test cases can be explored when code coverage-based testing is required.

As input intervals are attached to the desired input cells, test cases can be generated from the specified input intervals. This can be achieved as symbolic testing is also used for test case generation. However, the specification of expected behavior for the test cases which are automatically generated by the system requires further consideration.

- Other improvements can be made based on observations in an experimental study where real users are studied using the methodology. In order to see whether the idea works, a prototype tool is implemented which can be integrated on top of spreadsheet systems. To quantify the effectiveness of the methodology, it is important to study the effectiveness of the methodology by setting a controlled experiment which involves end-user subjects using the methodology and others without using the methodology. There are, however, some parameters which should be taken into consideration while carrying out such controlled experiments. Among them are the background of the users participating in the experiment and the type of problems given to the users during the experiment. Users should have similar background in their experience and the problems should be selected so as not to require a special domain knowledge. Similarity of the background of the subjects can be determined based on statistical measures.

  The first question that the experiment should answer is "Does it help in detecting faults in spreadsheet programs?". To answer this question, two groups of participants with similar backgrounds can be chosen for the experiment. While one group develops spreadsheets with the testing methodology integrated, the other group works without using the methodology. This helps to get information regarding the usefulness of the approach in enabling users to detect symptoms of faults in spreadsheet formulas. The second question that the experiment should answer is "How effective is the methodology?". This requires comparing the effectiveness with other approaches which are proposed for the same purpose. However, there are so far no tools which are designed for the purpose of testing spreadsheets with which comparison can be done.

Glossary

A **spreadsheet** is an n-dimensional matrix of cells where each cell is uniquely identified by n-coordinates. If $n = 2$, as in the standard case, a cell is uniquely identified by its row and column address. We will consider here only this standard case of tabular spreadsheets.

A **cell** is the atomic unit of a spreadsheet. Its content might be: (a) a **value** (numeric or textual literal) (b) a **formula** for computing some value by referring to the values of other cells (c) **empty** - a cell with neither a value nor a formula.

A **Formula** is a mathematical expression which might consist of cell references, operators, functions[1], and constant values. At least one cell reference is expected to be included in the computational expression of a formula. A formula yields exactly one result.

It is important to distinguish between the **tabular appearance** of a spreadsheet where all cells are shown to be either empty or contain a numeric or textual value and the **cell definitions** (constants or formulas) that lead to these values presented at the user interface.

Cell reference is a mechanism to refer to the values of other cells for the computation of the value of a given cell formula. Cell reference can be through name, absolute reference, and relative reference. A cell reference consists of two coordinate systems. In an **absolute reference**, the upper-left corner of the spreadsheet is considered as the origin. In a **relative reference**, reference to a cell is at a relative offset from the cell making the reference. Cells can also be referred by name. A name reference is

[1] A function is a built-in formula supplied by the spreadsheet system

just an alias for a cell address.

An **Input Cell** is a cell which contains a numeric value entered by the user.

A **Formula Cell** contains a formula definition for the computation of the value of the cell.

A **Spreadsheet Program** constitutes the set of cell definitions in conjunction with the particular location where these cell definitions appear. All computations are local to the Cells in which they are defined and free of side-effects.

A **Spreadsheet Instance** is a spreadsheet program where input cells have certain values. A spreadsheet program can be instantiated multiple times.

A **Spreadsheet Language** consists of a set of language constructs to describe the data flow (cell references) and the data manipulation (formulas) in a spreadsheet program.

A **Spreadsheet System** is an integrated environment where spreadsheet programs can be created, instantiated, and edited.

A **User** is any person who is not a professional programmer but uses spreadsheet systems for some computational purposes. User and spreadsheet programmer (developer) are synonymously used in this thesis.

The terms error and fault are used in the literature in different contexts. We use them based on the following definitions [47, 117].

An **Error** is a mental mistake made by a programmer that results in a fault.

A **Fault** is a bug in a program that can result in a failure. Fault, defect, and bug are synonymously used.

A **Failure** is a misbehavior of a program.

Bibliography

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Software Practice & Experience*, 23(6):589–616, June 1993.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault Localization using Execution Slices and Dataflow Tests. *Proceedings of the* 6^{th} *International Symposium on Software Reliability Engineering*, pages 143–151, October 1995.

[3] A. L. Ambler. Generalizing the sheet language paradigm. In T. Ichikawa, E. Jungert, and R. R. Korfhage, editors, *Visual Languages and Applications*, pages 327–347. Plenum Press, 1990.

[4] A. L. Ambler and M. Burnett. A Declarative Approach to Event-Handling in Visual Programming Languages. *IEEE Workshop on Visual Languages*, pages 34–40, September 1992.

[5] Y. Ayalew, M. Clermont, and R. T. Mittermeir. Detecting Errors in Spreadsheets. *Proceedings of EuSpRIG 2000 Symposium Spreadsheet Risks, Audit & Development Methods*, pages 51–62, July 2000.

[6] I. Bashir and A. L. Goel. *Testing Object-Oriented Software: Life Cycle Solutions.* Springer-Verlag New York, Inc., 1999.

[7] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, second edition, 1990.

[8] A. Berglas. The Spreadsheet Detective. *at http://www.uq.net.au/detective/*, 1997.

[9] B. Boehm and V. R. Basili. Software Defect Reduction Top 10 List. *IEEE Computer*, 34(1):135–137, January 2001.

[10] P. S. Brown and J. D. Gould. An Experimental Study of People Creating Spreadsheets. *ACM Transactions on Office Information Systems*, 5(3):258–272, July 1987.

[11] M. Burnett, J. Atwood, R. W. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming*, 2000. to appear.

[12] M. Burnett, M. J. Baker, C. Bohus, P. Carlson, P. van Zee, and S. Yang. The scaling-up problem for visual programming languages. *IEEE Computer*, 28(3):45–54, March 1995.

[13] M. Burnett, A. Sheretov, and G. Rothermel. Scaling Up a "What You See Is What You Test Methodology" to Spreadsheet Grids. *IEEE Symposium on Visual Languages*, pages 30–37, September 1999.

[14] R. J. Butler. Is This Spreadsheet a Tax Evader? How H.M. Customs & Excise Test Spreadsheet Applications. *Proceedings of the* 33^{rd} *Hawaii International Conference on System Sciences*, 2000.

[15] R. J. Casmir. Real programmers don't use spreadsheets. *SIGPLAN Notices*, 27(6):10–16, 1992.

[16] D. Chadwick, B. Knight, and K. Rajalingham. Quality Control in Spreadsheets: A Visual Approach using Color Codings to Reduce Errors in Formula. *Proceedings of the* 8^{th} *International Conference on Software Quality Management*, April 2000.

[17] D. Chadwick, J. Knight, and P. Clipsham. Information Integrity in End-user Systems. *Proceedings of the First Annual IFIP TC-11 Working Group 11.5 Working Conference on Integrity and Internal Control in Information Systems*, pages 273–292, 1997.

[18] D. Chadwick, K. Rajalingham, B. Knight, and D. Edwards. An Approach to the Teaching of Spreadsheets using Software Engineering Concepts. *Proceedings of the 4th International Conference on Software Process Improvement, Research, Education and Training*, pages 261 – 273, September 1999.

[19] S. K. Chang. Visual Languages: A Tutorial and Survey. In E. P. Glinert, editor, *Visual Programming Environments: Paradigms and Systems*, pages 7–17. IEEE Computer Society Press, 1990.

[20] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, 5(4):403–417, July 1979.

[21] Y. Chen and H. C. Chan. Visual Checking of Spreadsheets. *Proceedings of EuSpRIG 2000 Symposium Spreadsheet Risks, Audit & Development Methods*, pages 75–85, July 2000.

[22] E. H. Chi. *A Framework for Information Visualization Spreadsheets.* PhD thesis, University of Minnesota, March 1999.

[23] C. M. Chung and M. C. Lee. Object-oriented programming testing methodology. In E. J. Braude, editor, *Object Oriented Analysis, Design and Testing*, pages 179–186. IEEE, inc., 1998.

[24] P. D. Coward. A Review of Software Testing. *Information and Software Technology*, 30(3):189–198, April 1988.

[25] P. T. Cox and T. J. Smedley. Using Visual Programming to Extend the Power of Spreadsheet Computation. *Proceedings of the Wokshop on Advanced Visual Interfaces*, pages 153–161, 1994.

[26] J. S. Davis. Tools for spreadsheet auditing. *International Journal of Human-Computer Studies*, 45(4):429–442, 1996.

[27] DECISIONPRO. Product information of DECISIONPRO available at http://www.vanguardsw.com. 2000.

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

[29] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical Slicing for Software Fault Localization. *International Symposium on Software Testing and Analysis*, pages 121–134, January 1996.

[30] M. Dodge, C. Kinata, and C. Stinson. *Running Microsoft Excel 97*. Microsoft Press, 1997.

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

[32] G. Filby. *Spreadsheets in Science and Engineering: with 24 Tables*. Springer-verlag, 1998.

[33] T. Green. Noddy's Guide to Visual Programming. *British Computer Society, Human-Computer Interaction Group*, Autumn 1995.

[34] R. Gupta, M. J. Harrold, and M. L. Soffa. An Approach to Regression Testing using Slicing. *Journal of Software Testing, Verification, and Reliability*, 6(2):83–112, June 1996.

[35] D. Hamlet, B. Gifford, and B. Nikolik. Exploring Dataflow Testing of Arrays. *Proceedings of the International Conference on Software Engineering*, pages 118–128, May 1993.

[36] E. Hanson. *Global Optimization Using Interval Analysis*. Marcel Dekker, Inc., 1992.

[37] M. J. Harrold. Testing: A Roadmap. *Proceedings of the 22^{nd} International Conference on The future of Software engineering*, pages 61–72, June 2000.

[38] D. G. Hendry. Display-Based Problems in Spreadsheets: A Critical Incident and a Design Remedy. 11^{th} *International IEEE Symposium on Visual Languages*, pages 284–290, September 1995.

[39] D. G. Hendry and T. R. G. Green. Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model. *International Journal of Human-Computer Studies*, 40(6):1033–1065, 1994.

[40] W. Hetzel. *The Complete Guide To Software Testing.* QED Information Sciences, Inc., second edition, 1988.

[41] W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):13–23, July 1977.

[42] S. E. Hudson. User Interface Specification Using an Enhanced Spreadsheet Model. *ACM Transactions on Graphics*, pages 209–239, 1994.

[43] M. Hutchins, H. Foster, T. Goradia, , and T. Ostrand. Experiments on the Effectiveness of Dataflow-and Controlflow-Based Test Adequacy Criteria. *Proceedings of the* 16^{th} *International Conference on Software Engineering*, pages 191–200, May 1994.

[44] E. Hyvönen. Personal communication through e-mail. March 2000.

[45] E. Hyvönen and S. De Pascale. Interval computations on the spreadsheet. In R. B. Kearfott and V. Kreinovich, editors, *Applications of Interval Computations*, volume 3, pages 169–210. Kluwer Academic Publishers, January 1996.

[46] E. Hyvönen and S. De Pascale. A New Basis for Spreadsheet Computing: Interval Solver for Microsoft Excel. *Proceedings of the* 11^{th} *Innovative Applications of Artificial Intelligence*, pages 799–806, July 1999.

[47] IEEE. *IEEE Software Engineering Standards Collection.* IEEE Computer Society Press, 1991.

[48] T. Igarashi, J. D. Mackinlay, B. W. Chang, and P. T. Zellweger. Fluid Visualization of Spreadsheet Structures. *IEEE Symposium on Visual Languages*, pages 118–125, September 1998.

[49] T. Isakowitz, S. Schocken, and H. C. Lucas. Toward a Logical/Physical Theory of Spreadsheet Modeling. *ACM Transactions on Information Systems*, 13(1):1–37, January 1995.

[50] P. Jalote. *An Integrated Approach to Software Engineering.* Springer-Verlag New York,Inc., second edition, 1997.

[51] D. Janvrin and J. Morrison. Using a structured design approach to reduce risks in end user spreadsheet development. *Information & Management*, 37:1–12, 2000.

[52] A. Kaufmann and M. M. Gupta. *Introduction to Fuzzy Arithmetic: Theory and Applications.* Van Nostrand Reinhold, New York, 1991.

[53] B. Knight, D. Chadwick, and K. Rajalingham. A Structured Methodology for Spreadsheet Modeling. *Proceedings of EuSpRIG 2000 Symposium Spreadsheet Risks, Audit & Development Methods*, pages 43–50, July 2000.

[54] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[55] B. Korel and J. Rilling. Application of Dynamic Slicing in Program Debugging. *Proceedings of the 3^{rd} International Workshop on Automatic Debugging*, pages 43–57, May 1997.

[56] J. Kreie, T. P. Cronan, J. Pendley, and J. S. Renwick. Applications development by end-users: can quality be improved? *Decisions Support Systems*, 29:143–152, 2000.

[57] S. E. Kruck. Towards a Theory of Spreadsheet Accuracy: An Empirical Study. *Proceedings of the 4^{th} Americas Conference on Information Systems*, August 1998.

[58] D. C. Kung, P. Hsia, and J. Gao. *Testing Object-Oriented Software.* IEEE Computer Society, 1998.

[59] J. W. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, May 1983.

[60] C. Lewis. NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery. In E. P. Glinert, editor, *Visual Programming Environments: Paradigms and Systems*, pages 526–546. IEEE Computer Society Press, 1990.

[61] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. *Proceedings of the* 2^{nd} *International Conference on Computers and Applications*, pages 877–883, June 1987.

[62] J. Martin. *Application Development Without Programmers.* Prentice-Hall, inc., 1982.

[63] D. Mather. A framework for building spreadsheet based decision models. *Journal of Operational Research Society*, 50:70–74, 1999.

[64] P. M. Maurer. Generating testing data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, July 1990.

[65] J. McDonald and P. Strooper. Translating Object-Z Specifications to Passive Test Oracles. 2^{nd} *International Conference on Formal Engineering Methods*, pages 165–174, 1998.

[66] Sun Microsystems. Interval Arithmetic Programming Reference. http://docs.sun.com, 2000.

[67] R. M. Miller. *Computer-Aided Financial Analysis.* Addison-Wesley, 1990.

[68] R. E. Moore. *Interval Analysis.* Prentice-Hall, Inc., New Jersey, 1966.

[69] D. J. Mosley. *The Handbook of MIS Application Software Testing: methods, techniques, and tools for assuring quality through testing.* Prentice-Hall Yourdon Press, New Jersey, 1993.

[70] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.

[71] G. J. Myers. *The Art of Software Testing.* Wiley-Interscience, 1979.

[72] B. A. Nardi. *A small matter of programming : perspectives on end user computing.* The MIT Press, 1993.

[73] B. A. Nardi and J. R. Miller. The Spreadsheet Interface: A Basis for End User Programming. Technical Report HPL-90-08, Hewlett-Packard Software Technology Laboratory, March 1990.

[74] B. A. Nardi and J. R. Miller. Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *International Journal of Man-Machine Studies*, 34:161–184, 1991.

[75] S. Ntafos. An Evaluation of Required Element Testing Strategies. *Proceedings of Seventh International Conference on Software Engineering*, pages 250–256, March 1984.

[76] A. J. Offutt. A Practical System for Mutation Testing: Help for the Common Programmer. 12^{th} *International Conference on Testing Computer Software*, pages 99–109, June 1995.

[77] A. J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. *Proceedings of Second International Conference on the Unified Modeling Language*, pages 416–429, October 1999.

[78] A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Approach to Test Data Generation. *Software-Practice and Experience*, 29(2):167–193, January 1999.

[79] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An Experimental Evaluation of Data Flow and Mutation Testing. *Software Practice and Experience*, 26(2):165–176, February 1966.

[80] J. Paine. Model Master: An Object-Oriented Spreadsheet Front End. *Computer Aided Learning - Using technology in economics and business education,*, 1997.

[81] R. R. Panko. What we Know About Spreadsheet Errors. *Journal of End User Computing: Special issue on Scaling Up End User Development*, 10(2):15–21, 1998.

[82] R. R. Panko. Applying code inspection to spreadsheet testing. *Journal of Management Information Systems*, 16(2):159–176, 1999.

[83] R. R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. *Proceedings of EuSpRIG Symposium Spreadsheet Risks, Audit & Development Methods*, pages 7–14, July 2000.

[84] R. R. Panko. Two Corpuses of Spreadsheet Errors. *Proceedings of the* 33^{rd} *Hawaii International Conference on System Sciences*, 2000.

[85] R. R. Panko and R. P. Halverson. Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks. *Proceedings of the* 29^{th} *Hawaii International Conference on System Sciences*, pages 326–335, January 1996.

[86] R. R. Panko and R. P. Halverson. Are two heads better than one? (at reducing errors in spreadsheet modeling). *Office Systems Research Journal*, 15(1):21–32, 1997.

[87] R. R. Panko and R. H. Sprague. Hitting the wall: Errors in developing and code inspecting a 'simple' spreadsheet model. *Decision Support Systems*, 22(4):337–353, April 1998.

[88] W. Perry. *Effective Methods for Software Testing.* John Wiley & Sons,Inc., U.S.A, 1995.

[89] D. K. Peters and D. L. Parnas. Generating a test oracle from program documentation-work in progress. *Proceedings of the 1994 international symposium on software testing and analysis(ISSTA)*, pages 58–65, August 1994.

[90] D. K. Peters and D. L. Parnas. Using Test Oracles Generated from Program Documentation. *Proceedings of the* 3^{rd} *IEEE Transactions on Software Engineering*, 24(3):161–173, March 1998.

[91] R. S. Pressman. *Software Engineering: A practitioner's Approach.* McGRAW-HILL, third edition, 1992.

[92] K. Rajaligham, D. Chadwick, and B. Knight. Classification of Spreadsheet Errors. *Proceedings of EuSpRIG 2000 Symposium Spreadsheet Risks, Audit & Development Methods*, pages 23–34, July 2000.

[93] K. Rajalingham and D. Chadwick. Efficient Methods for Checking Integrity: Integrity Control of Spreadsheets, Organisation & Tools. *The* 2^{nd} *Annual IFIP TC-11 WG.5 Working Conference on Integrity and Internal Control in Information Systems*, 1998.

[94] K. Rajalingham, D. Chadwick, B. Knight, and D. Edwards. Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development. *Proceedings of the* 33^{rd} *Hawaii International Conference on System Sciences*, 2000.

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

[96] J. Reichwein, G. Rothermel, and M. Burnett. Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging. *Proceedings of the* 2^{nd} *Conference on Domain Specific Languages*, pages 25–38, October 1999.

[97] B. Ronen, M. A. Palley, and H. C. Lucas. Spreadsheet Analysis and Design. *Communications of the ACM*, 32(1):84–92, 1989.

[98] G. Rothermel and M. J. Harrold. A Framework for Evaluating Regression Test Selection Techniques. *Proccedings of the* 16^{th} *International Conference on Software Engineering*, pages 201–210, May 1994.

[99] G. Rothermel, L. Li, and M. Burnett. Testing Strategies for Form-Based Programs. *Proceedings of the* 8^{th} *International Symposium on Software Reliabilty Engineering*, pages 96–107, November 1997.

[100] G. Rothermel, L. Li, C. DuPuis, , and M. Burnett. What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs. *Proceedings*

of the 20^{th} *International Conference on Software Engineering*, pages 198–207, April 1998.

[101] P. Saariluoma and J. Sajaniemi. Transforming Verbal Descriptions into Mathematical Formulas in Spreadsheet Calculation. *International Journal of Human-Computer Studies*, 41(6):915–948, 1994.

[102] J. Sajaniemi. Modelling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *Journal of Visual Languages and Computing*, 11(1):49–82, June 2000.

[103] J. Sajaniemi and J. Pekkanen. An empirical analysis of spreadsheet calculation. *Software Practice and Experience*, 18:583–596, 1988.

[104] J. Sajaniemi, M. Tukiainen, and J. Vaisanen. Goals and Plans in Spreadsheet Calculation. Technical Report Report A-1999-1, Department of Computer Science, University of Joensuu, 1999.

[105] R. Scoville. Spreadsheets. In E. Knorr, editor, *The PC Bible*, page 403. Peachpit Press, 1994.

[106] H. Shiozawa, K. Okada, and Y. Matsushita. 3D Interactive Visualization for Inter-Cell Dependencies of Spreadsheets. *Proceedings of the 1999 IEEE Symposium on Information Visualization*, pages 79–82, October 1999.

[107] N. C. Shu. Visual programming languages: A perspective and a dimensional analysis. In E. P. Glinert, editor, *Visual Programming Environments: Paradigms and Systems*, pages 41–58. IEEE Computer Society Press, 1990.

[108] J. Smith. Spreadsheet sales to double by 1993. *Lotus*, 6:12, 1990.

[109] I. Sommerville. *Software Engineering.* Addison-Wesley, fourth edition, 1992.

[110] M. Stadelman. A Spreadsheet Based on Constraints. *Proceedings of the* 6^{th} *Annual ACM Symposium on User Interface Software and Technology*, pages 217–224, 1993.

[111] K. C. Tai. Predicate-Based Test Generation for Computer Programs. *Proceedings of the* 15^{th} *International Conference on Software Engineering*, pages 267–276, May 1993.

[112] F. Tip. A Survey of Program Slicing Techniques. Technical Report CS-R9438, Computer Science/Department of Software Technology, CWI, P.O.Box 94079, 1090 GB Amsterdam, The Netherlands, 1994.

[113] M. Tukiainen. ASSET: A Structured Spreadsheet Calculation System. *Machine-Mediated Learning*, 5(2):63–76, 1996.

[114] M. Tukiainen. Uncovering Effects of Programming Paradigms: Errors in Two Spreadsheet Systems. 12^{th} *Workshop of the Psychology of Programming Interest Group*, pages 247–266, April 2000.

[115] M. Tukiainen and J. Sajaniemi. Spreadsheet Goal and Plan Catalog: Additive and Multiplicative Computational Goals and Plans in Spreadsheet Calculation. Technical Report A-1996-4, Department of Computer Science, University of Joensuu, 1996.

[116] J. Tupper. Graphing Equations With Generalized Interval Arithmetic. Master's thesis, University of Toronto, Department of Computer Science, 1996.

[117] J. M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons, Inc., 1998.

[118] J. M. Voas, L. Morell, and K. W. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–58, March 1991.

[119] A. H. Watson and T. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Technical Report NIST Special Publication 500-235, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-0001, USA, August 1996.

[120] N. P. Wilde. A WYSIWYC(What You See IS What You Compute) Spreadsheet. *Proceedings of the 1993 Symposium on Visual Languages*, pages 72–76, August 1993.

[121] A. G. Yoder and D. L. Cohn. Architectural Issues in Spreadsheet Languages. *International Conference on Programming Languages and System Architectures*, March 1993.

[122] A. G. Yoder and D. L. Cohn. Observations on Spreadsheet Languages, Intension and Dataflow. Technical Report TR-94-22, Distributed Computing Research Lab, University of Notre Dame, 1994.

[123] A. G. Yoder and D. L. Cohn. Real Spreadsheets for Real Programmers. *International Conference on Computer Languages*, pages 20–30, May 1994.

[124] A. G. Yoder and D. L. Cohn. Domain-Specific and General-Purpose Aspects of Spreadsheet Languages. *First ACM SIGPLAN Workshop on Domain-Specific Languages*, pages 37–47, 1997.