AN ABSTRACT OF THE DISSERTATION OF

Robin Abraham for the degree of Doctor of Philosophy in Computer Science presented on April 19, 2007.

Title: End-User Software Engineering in the Spreadsheet Paradigm

Abstract approved: _____

Martin Erwig

Spreadsheets are among the most widely used end-user programming systems. Unfortunately, there is a high incidence of errors in end-user spreadsheets, and some of these errors have high impact. In this dissertation, we describe techniques we have developed to help end users develop safer spreadsheets. As part of our research, we have examined typical software development activities like programming, testing, and debugging from an end-user software engineering perspective, and developed approaches to support them better. To achieve this goal, we have drawn extensively from research in the areas of software engineering and programming languages to develop techniques for the following.

1. Prevention of reference, range, and type errors by generating spreadsheets from user-defined templates.
2. Detection of unit errors by carrying out automatic consistency checking of spreadsheet formulas using labels within spreadsheets.

3. Detection of faults within spreadsheets by helping end users test better using constraint-based automatic test-case generation.
4. Correction of identified faults within spreadsheet formulas using a spreadsheet debugger that exploits the user's expectations about the output of their spreadsheets.

©Copyright by Robin Abraham
April 19, 2007
All Rights Reserved

End-User Software Engineering in the Spreadsheet Paradigm

by

Robin Abraham

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented April 19, 2007
Commencement June 2007

Doctor of Philosophy dissertation of Robin Abraham presented on April 19, 2007.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Robin Abraham, Author

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Martin Erwig for his patience and for being incredibly generous with his time.

I would also like to thank members of the programming languages and Forms/3 research groups at Oregon State University, and faculty and student collaborators within the EUSES consortium for the many valuable discussions and inputs pertaining to my research.

TABLE OF CONTENTS

Page

1 Introduction 1

2 A Formal Overview of Spreadsheet Programs 6
- 2.1 Spreadsheet Programs . . . 6
- 2.2 Testing Spreadsheets . . . 10
- 2.3 Classifications of Spreadsheet Cells . . . 11

3 Literature Review 14
- 3.1 Spreadsheet Systems . . . 19
- 3.2 Specification-Based Approaches . . . 22
- 3.3 Static Checking . . . 25
  - 3.3.1 Unit Checking . . . 26
  - 3.3.2 XeLda . . . 30
  - 3.3.3 SLATE . . . 33
- 3.4 Testing . . . 34
  - 3.4.1 "What You See Is What You Test" (WYSIWYT) . . . 36
  - 3.4.2 "Help Me Test" (HMT) . . . 38
- 3.5 Debugging . . . 39

4 Specifications-Based Approach to Spreadsheet Correctness 42
- 4.1 A Visual Specification Language for Spreadsheet Templates . . . 47
- 4.2 Generating Correct Spreadsheets using Gencel . . . 52
- 4.3 Automatic Inference of Spreadsheet Templates . . . 55
- 4.4 Evaluating the Quality of Inferred Templates . . . 63
  - 4.4.1 Experiment Design . . . 65
  - 4.4.2 Threats to Validity . . . 68
  - 4.4.3 Results . . . 69
- 4.5 Error Detection Through Template Inference . . . 74
- 4.6 Conclusions and Future Work . . . 75

TABLE OF CONTENTS (Continued)

Page

5 Automatic Consistency Checking for Spreadsheets 77

5.1 Unit Inference . . . 79
5.1.1 Unit Expressions . . . 79
5.1.2 A Formal Unit System . . . 82
5.1.3 An Example for Unit Checking . . . 85

5.2 Header Inference . . . 88
5.2.1 Header-inference Framework . . . 88
5.2.2 Header Assignment . . . 91
5.2.3 Optimizing Header Inference . . . 96

5.3 Preliminary Evaluation . . . 98

5.4 Think-Aloud Study . . . 100
5.4.1 Participants . . . 102
5.4.2 Procedure . . . 103
5.4.3 Research Questions . . . 105
5.4.4 Observations . . . 106
5.4.5 Discussion . . . 111
5.4.6 Results . . . 113

5.5 Conclusions and Future Work . . . 115

6 Automatic Test-Case Generation 117

6.1 Testing With AutoTest . . . 118

6.2 Definition-Use Coverage . . . 122
6.2.1 Constraint Trees . . . 123
6.2.2 Definition-Use Pairs . . . 125
6.2.3 Testing Definition-Use Pairs . . . 126

6.3 Evaluation . . . 130

6.4 Conclusions and Future Work . . . 133

7 Debugging End-User Spreadsheets 134

7.1 Debugging With Change Suggestions . . . 136

7.2 Change Inference . . . 139

7.3 Ranking Heuristics . . . 144

7.4 Conclusions and Future Work . . . 147

TABLE OF CONTENTS (Continued)

Page

8 Mutation Operators for Spreadsheets 149
8.1 Mutation Testing . . . 150
8.2 Mutation Operators for Spreadsheets . . . 151
8.3 Mutation Coverage of Du-Adequate Test Suites . . . 158
8.3.1 Experiment Setup . . . 159
8.3.2 Results . . . 160
8.3.3 Analysis of Surviving Mutants . . . 163
8.3.4 Mutation Testing With Higher-Order Mutants . . . 168
8.3.5 Extending AutoTest . . . 169
8.3.6 Discussion . . . 173
8.4 Evaluation of GoalDebug . . . 173
8.4.1 Experiment Setup . . . 174
8.4.2 Preliminary Evaluation . . . 177
8.4.3 Evaluation of the Extended System . . . 178
8.4.4 Discussion . . . 180
8.5 Evaluation of Combined Reasoning Approaches . . . 181
8.5.1 Experiment Setup . . . 184
8.5.2 Results . . . 185
8.5.3 Discussion . . . 185
8.6 Conclusions and Future Work . . . 187

9 Conclusions 189

Bibliography 192

LIST OF FIGURES

Figure		Page
3.1	ClassSheets specification (a) and the corresponding UML class diagram (b) for accounting spreadsheets	24
3.2	Unit checking	27
3.3	Specifying header structure in unit checker	28
3.4	Unit inference example	29
3.5	Specifying headers and dimensions in XeLda	30
3.6	Dimension checking in Xelda	32
3.7	Cell annotations in Slate	33
3.8	Fault localization feedback in WYSIWYT	36
3.9	Arrows showing definitions and uses in Forms/3	37
3.10	Debugging using Whyline	40
4.1	"Traditional" approach to spreadsheet development.	44
4.2	Proposed approach to spreadsheet development.	45
4.3	ViTSL/Gencel approach to spreadsheet development.	46
4.4	ViTSL template for a summation column	47
4.5	Grade sheet template in ViTSL.	48
4.6	Templates.	50
4.7	Grade sheet in Gencel.	53
4.8	Template inferred from grade sheet.	57
4.9	Sample grade sheet.	59
4.10	Overview of evaluation setup	66
4.11	Inferred templates	69
4.12	Score comparisons	70

LIST OF FIGURES (Continued)

Figure		Page
4.13	Subject behavior on tasks	72
4.14	Template inference using spreadsheet types.	75
5.1	Harvest spreadsheet	78
5.2	Unit inference rules.	83
5.3	Unit transformations.	84
5.4	Well-formed units.	84
5.5	Unit equality.	86
5.6	Inferred headers.	92
5.7	Car exports.	95
5.8	Unsatisfactory header inference.	97
5.9	Grade sheet with unit errors.	103
5.10	Enrollment sheet with unit errors.	104
6.1	Office supplies spreadsheet	119
6.2	Automatically generated test cases for B9	120
6.3	Stages of test-case generation	124
7.1	Gradebook spreadsheet	137
7.2	Specifying expected output for a cell	137
7.3	Change suggestions	138
7.4	Applying suggestion	138
7.5	Change Inference	140
8.1	Setup for evaluating change suggestions	174
8.2	Comparison of coverage%	181
8.3	Cumulative coverage of ranking heuristics	182

LIST OF FIGURES (Continued)

Figure Page

8.4 Combining reasoning from UCheck and WYSIWYT 185

LIST OF TABLES

Table Page

2.1 Cell roles 13

4.1 Toolbar options for constructing ViTSL templates 49

4.2 Scoring criteria for templates 67

4.3 κ values for pairs of graders 70

6.1 Ultimate effectiveness scores of the different techniques per spreadsheet. 131

6.2 Response times (in seconds) of the different techniques per spreadsheet. 132

7.1 User mistakes during debugging 135

8.1 Subset of mutation operators for FORTRAN 153

8.2 Correspondence of constructs in general-purpose programming languages and in spreadsheets 153

8.3 Mutation operators for spreadsheets 154

8.4 Mutation adequacy (running first test per definition-use pair) 161

8.5 Mutation adequacy (running single randomly selected test per definition-use pair) averaged over 5 runs) 162

8.6 Mutation adequacy (running all tests) 163

8.7 Comparison of number of test cases against mutation coverage 164

8.8 Mutation operators and live mutants (A: #alive mutants, T: Total #mutants seeded,%: Survival percentage) 164

8.9 Mutation adequacy of AutoTest test suites with second-order mutants . 168

8.10 Sheets used in the evaluation 176

8.11 Original version of GoalDebug's effectiveness at correcting mutations . 178

8.12 GoalDebug's effectiveness at correcting mutations after enhancements . 180

8.13 Visual Effectiveness scores with original mapping 186

8.14 Visual Effectiveness scores with threshold mapping 186

LIST OF TABLES (Continued)

Table Page

8.15 Observed failures vs. reasoning 187

DEDICATION

To my parents, Abraham and Omana, and my brother, Roy, for their encouragement and support.

END-USER SOFTWARE ENGINEERING IN THE SPREADSHEET PARADIGM

Chapter 1 – Introduction

In the U.S. alone the number of end-user programmers is estimated to be 11 million, compared to only 2.75 million other, professional programmers [106]. End users in the context of our work are people who are not trained at programming but still do some amount of programming as part of their work. They are not really interested in investing the time and effort to learn concepts in computer science or software engineering. They simply want to get their real work done. Some popular end user programming environments are web authoring tools [40, 54], graphical languages for creating educational simulations [14, 36], and spreadsheet systems [57, 60, 79, 86]. With the increase in number and availability of end-user programming environments, more and more end users are developing software.

> *The ranks of "sorcerer's apprentice" user-programmers will also swell rapidly, giving many who have little training or expertise in how to avoid or detect high-risk defects tremendous power to create high-risk defects [24].*

Training the so-called "sorcerer's apprentice" user programmers so that they have the expertise to avoid or detect high-risk defects is obviously not a feasible option. Instead, with support from the National Science Foundation (under the grant ITR-0325273) and the EUSES consortium [46], we have focused our efforts on a new way of thinking about

how end users create software and how best to support their activities. Specifically, we would like to address the following research question:

> *Can the benefits of sound software engineering concepts be brought within the grasp of end users in order to help them develop effective software?*

We have chosen to focus our research on the spreadsheet paradigm since studies have shown that spreadsheets are among the most widely used end-user programming environments [91]. Many studies have also shown that spreadsheets contain errors at alarmingly high rates [25, 75, 115, 117]. Some studies estimate that 90% of real-world spreadsheets have non-trivial errors in them [98]. In many instances, spreadsheet errors have resulted in huge losses to the organizations concerned [37, 56, 58, 107]. Many of these "horror stories" that have appeared in the media have been documented at [47].

The flexibility afforded by most spreadsheet systems make it very easy to get started working with them. The user can run a spreadsheet system like Microsoft Excel, start a new spreadsheet and enter a number in a cell. The sheet can now be saved, and the user has created a brand new (albeit trivial) spreadsheet. The user could, with relative ease, also use the spreadsheet to store data. The free-form grid layout and immediate visual feedback make it a good tool for simple simulations and modeling. The flexibility offered by spreadsheet systems also makes it very easy to create erroneous spreadsheets or introduce errors in existing ones. Such risks are unlikely to decrease since studies have shown that end users are creating more and more complex spreadsheets [109], with typical corporate spreadsheets doubling in size and formula content every three years [122].

The key design goal for approaches to support end-user software engineering is to provide support without being intrusive. Researchers have to overcome the following challenges that are unique to end-user software engineering [102]:

- We cannot expect the user to have any prior knowledge of or even interest in learning software engineering techniques and practices.
- High interactivity is a typical feature of end-user programming environments. Therefore, any tool or approach aimed at supporting programming in these environments must provide incremental and immediate feedback.
- End users may not have the knowledge or understanding to guide the system. Therefore, the techniques should work with whatever information is available and require little or no help from the user.
- There is ample empirical evidence which indicates that end users make errors during program construction [71], testing [103], and debugging [72]. Any tool or approach aimed at supporting these activities should be tolerant of user errors and flexible enough to accommodate different learning styles and strategies.
- End users will not want to use a technique unless the perceived benefits outweigh the perceived costs and risks [22]. Therefore, in addition to being effective and dependable, any tool aimed at end users should also give them obvious evidence of its benefits in order to coax them to continue using it.

In order to facilitate software engineering by end-users, we have worked on approaches to support typical software engineering activities like design, development, testing, and debugging within the spreadsheet paradigm.

In Chapter 2 we present the formal notation used in the remainder of the dissertation. The rest of the chapters are mostly self contained. Related work is described in Chapter 3. In Chapter 4, we describe an approach we have developed that allows end users to create spreadsheet templates and then generate spreadsheets that conform to the template.

Even though there are many advantages in using explicit templates to generate spreadsheets, the approach is not as flexible as working directly with a spreadsheet. In Chapter 5, we describe an automatic consistency checker that exploits the labels entered by users in their spreadsheets to detect errors in spreadsheet formulas. This approach is flexible and inexpensive.

Automatic consistency checkers do not detect all possible errors within spreadsheets. Therefore, it is also important to test programs to expose faults. Since it is unreasonable to expect end users to carry out thorough testing of their spreadsheets, we have developed an automatic test-case generator, described in Chapter 6, that generates *definition-use adequate* test suites for spreadsheets.

Exposing faults through testing is only the first step in correcting errors in spreadsheets. In Chapter 7, we describe a new approach we have developed to help end users debug their spreadsheets. The approach, called "goal-directed debugging", allows the user to specify expected output for cells that exhibit failures. The user expectations are used to generate a set of suggested program changes that would result in the expected output being computed in those cells.

Following work done on mutation testing in the domain of general-purpose programming languages, we have developed a collection of mutation operators for spreadsheets.

The operators can be used to evaluate the quality of test suites and also for seeding errors in spreadsheets being used for empirical evaluations. We describe the operators and a few of the empirical evaluations we have used them for in Chapter 8. We present conclusions and future directions for research in Chapter 9.

Chapter 2 – A Formal Overview of Spreadsheet Programs

This chapter presents formal notations and general concepts used in the descriptions in later chapters. A formal model of spreadsheet programs is described in the next section since the notions pertaining to programs in general-purpose programming languages have to be slightly adjusted for spreadsheets. Unlike programs in general-purpose programming languages, spreadsheets contain both the program component and the inputs to the program. We use the model of spreadsheet programs described in the next section to develop a model of test cases and test suites for spreadsheets in Section 2.2. Many of the approaches described in the following chapters exploit *similarities* between spreadsheet formulas, both for the inference of regions and for the generation of change suggestions (and their ranking on the basis of likelihood). In Section 2.3, we describe the classifications used for "measuring" similarities between spreadsheet formulas.

2.1 Spreadsheet Programs

A spreadsheet program can be considered as given by a set of formulas that are indexed by cell addresses taken from the set $A = \mathbb{N} \times \mathbb{N}$. A set of addresses $s \subseteq A$ is called a *shape*. Shapes can be derived from references of a formula or from the domain of a group of cells and provide structural information that can be exploited in different ways.

We assume a set F that contains all possible formulas. Cell formulas ($f \in F$) are

either plain values $v \in V$, references to other cells (given by address $a \in A$), or operations (ψ) applied to one or more argument formulas.

$$f \in F \quad ::= \quad v \mid a \mid \psi(f, \ldots, f)$$

Formulas can evaluate to plain values $v \in V$ or to a special error value (ε). V contains numbers, strings, and a special blank value ($\sqcup$). ε has been introduced since formulas sometimes evaluate to errors (for example, circular references). Blank values represent empty cells. Operations include binary operations, aggregations, and, in particular, a branching construct $\mathsf{IF}(f,f,f)$.

A cell (c) within the spreadsheet is a pair (a, f) composed of the cell address a and formula f (or value). A spreadsheet is given by a partial function $S : A \to F$ mapping cell addresses to formulas (and values). $S(a)$ returns the formula (or value) stored at address a within the spreadsheet S. Similarly, $C_S(i, j)$ returns the formula (or value) stored within the cell at the intersection of column i and row j. Therefore, for an address $a = (i, j)$, $S(a) \equiv C_S(i, j)$. In discussions pertaining to regions within spreadsheets, we use $S\lceil i \rceil$ to denote the ith column within the spreadsheet S.

The semantics of spreadsheets is given by a function $[\![\ ]\!] : F \times S \to V \cup \{\varepsilon\}$. The evaluation of a formula f in the context of a spreadsheet S is represented by $[\![f]\!]_S$. The result is a value $v \in V$ or ε.

We define a function $\sigma : F \to 2^A$ that, given a formula $f \in F$, computes the addresses

of the cells it references. That is, $\sigma(f)$ is the shape of f, and can be defined as follows.

$$\begin{aligned}
\sigma(v) &= \varnothing \\
\sigma(a) &= \{a\} \\
\sigma(\psi(f_1,\ldots,f_k)) &= \sigma(f_1)\cup\ldots\cup\sigma(f_k)
\end{aligned}$$

The function σ can be extended to work on cells and cell addresses by $\sigma(c) = \sigma(a,f) = \sigma(f)$ and $\sigma(a) = \sigma(S(a))$. Since some of the cells within $\sigma(f)$ might themselves contain formulas with references, we define a related function $\sigma^*_S : S \times F \rightarrow 2^A$ that transitively chases references to determine all the input cells (that contain only values and not formulas) for the given formula. The definition of σ^*_S is identical to that of σ except for the following case:

$$\sigma^*_S(a) = \begin{cases} \{a\} & \text{if } S(a) \in V \\ \sigma^*_S(S(a)) & \text{otherwise} \end{cases}$$

Like σ, σ^*_S can also be extended to work on cells and addresses. The cells addressed by $\sigma^*_S(c)$ are also called c's *input cells*.

We can also define a function σ'_S that transitively builds a set of references that are upstream from a given cell in the spreadsheet program.

$$\begin{aligned}
\sigma'_S(a,v) &= \varnothing \\
\sigma'_S(a,\uparrow a') &= \{a\}\cup\sigma'_S(a',S(a')) \\
\sigma'_S(a,\psi(f_1,\ldots,f_k)) &= \sigma'_S(a,f_1)\cup\ldots\cup\sigma'_S(a,f_k)
\end{aligned}$$

We can use σ'_S to define the function ϕ, which for a given cell address, gives the set of all cell addresses that reference it directly or indirectly.

$$\phi(a) = \{a' \mid a \in \sigma(a', S(a'))\}$$

Two of the systems described in later chapters infer formula changes within spreadsheets. The system described in Chapter 7 infers changes which are recommended to the user, and the system described in Chapter 8 infers changes which are applied to the spreadsheet to generate *mutant* spreadsheets. In either case, it is important to ensure that the reference changes that are applied do not introduce cycles within the spreadsheet. The predicate ρ which checks if the introduction of a reference $\uparrow a'$ to the formula in cell a would introduce a cycle in spreadsheet S can be defined using the function ϕ as follows.

$$\rho(S, a, \uparrow a') = a' \notin \phi(a)$$

To extend the view of programs and their outputs to spreadsheets, we can observe that a spreadsheet contains a program together with the corresponding input. More precisely, the *program part* (P_S) of a spreadsheet S is given by all the cells that contain non-trivial formulas. That is, $P_S = \{(a, f) \in S \mid \sigma(f) \neq \varnothing\}$. This definition ignores formulas like $2+3$ and does not consider them a part of the spreadsheet program since they always compute the same output and can effectively be replaced with a constant. Correspondingly, the *input* component (D_S) of a spreadsheet S is given by all the cells that contain values (and locally computable formulas). That is, $D_S = \{(a, f) \in S \mid \sigma(f) = \varnothing\}$. Given the above definitions of P_S and D_S, a spreadsheet is a partition of program

and input. That is, we have $S = P_S \cup D_S$ and $P_S \cap D_S = \varnothing$.

2.2 Testing Spreadsheets

Given the above definitions, we are in a position to define test cases formally in the context of spreadsheets. A *test case* for a cell (a, f) is a pair (I, v) consisting of values for all the input cells that are in the *backward slice*[1] of f, given by I, and the expected output for f, given by $v \in V$. Since the input values are tied to cell addresses, the input part of the test case is itself a spreadsheet that contains only values, that is $I : A \to V$. However, not any I will do; we require that the domain of I matches f's inputs, that is $dom(I) = \sigma^*_S(f)$. We use σ^*_S in this definition, and not σ, since the input values are given by the data cells $(a, f) \in D_S$.

Running a formula f on a test case means to evaluate f in the context of I. We define that a formula f *passes* a test case $t = (i, v)$ if $[\![f]\!]_I = v$. Otherwise, f fails the test case. We can extend this notion to a cell and say that a cell $c = (a, f)$ passes (or fails) a test case t if f passes (or fails) t.

We need to distinguish between the notions of testing an individual formula within a spreadsheet and testing the entire spreadsheet itself. First, a *test set* for a cell c is a set of tests $T = \{t_1, \ldots, t_n\}$ such that each t_i is a test case for c. Second, a *test suite* for a spreadsheet S is a collection of test sets $\mathcal{T}_S = \{(a, T_a) \mid a \in dom(P_S)\}$ such that T_a is a test set for the cell with address a. A test suite $\mathcal{T}_S$ in which each test set T_a contains just a single test case (that is, $|T_a| = 1$) is also called a *test sheet* for S. Running a formula f

[1] The backward slice of f consists of all the cells that affect f.

on a test set T means to run f on every $t \in T$. Running a spreadsheet S on a test suite $\mathcal{T}_S$ means to run for every $(a, f) \in P_S$, the formula f on the test set $\mathcal{T}_S(a)$.

A formula f (or cell c) *passes* a test set T if and only if f (or c) passes every test case $t_i \in T$. Likewise, a spreadsheet S *passes* a test suite $\mathcal{T}_S$ if for every $(a, f) \in P_S$, f passes $\mathcal{T}_S(a)$. Otherwise, S *fails* the test suite $\mathcal{T}_S$.

2.3 Classifications of Spreadsheet Cells

Cell classifications are important for identifying homogenous regions within spreadsheets. Classifications of cell formulas have to take into account the reference scheme used. Excel allows two reference schemes in cells.

1. In the *A1-style* referencing scheme, relative references are of the form A2 (both the row and column change when the reference is copied to a new cell) and absolute references are of the form A$3 (the row number remains unchanged if the reference is copied to a new location), $A3 (the column number remains unchanged if the reference is copied to a new location), or A3 (both the column and rows remain unchanged if the reference is copied to a new location).
2. In the *R1C1-style*, a reference B3 in cell C3, for example, would be represented as RC[-1]—reference the cell in this row and one column to the left of this one. Along similar lines, a formula =B3/B$2 in cell C3 could be represented as =RC[-1]/R2C[-1] in the *R1C1* style.

The classifications described below are not orthogonal. For example, *computationally dead cells* under formula similarities is the same as *labels* under classification by

cell roles.

Formula similarities: Research aimed at identifying similarities between cell formulas has led to the formulation of the following *equivalence classes* [67, 80, 105]:

1. Node equivalence classes:
   (a) Copy equivalence: Two cells are *copy equivalent* if their formulas are identical (references are compared in the *R1C1* notation). This relationship holds even when the cells are not spatially adjacent.
   (b) Logical equivalence: Two cells are *logically equivalent* if their formulas differ only in constant values and absolute references.
   (c) Structural equivalence: Two cells are *structurally equivalent* if their formulas contain the same operations in the same order.
2. Link equivalence classes:
   (a) Source equivalence: Two cells are *source equivalent* if all their relative references have equal coordinates. Absolute references are not considered in this case.
   (b) Sink equivalence: Two cells are *sink equivalent* if they are used by two other cells that are source equivalent.
   (c) Aggregation equivalent: Two cells are *aggregation equivalent* if there is some other cell that references them in an aggregation formula.
   (d) Data equivalence: Two cells are *data equivalent* if they reference identical cells.

	formula	data
used	intermediate	input
not used	output	label

Table 2.1: Cell roles

(e) Computationally dead cells: The cells in this class are not referenced by other cells and they do not have references to other cells.

Cell roles: Spreadsheet cells can be classified on the basis of the roles they play, as shown in Table 2.1. We say that a cell c is *used* by another cell c' if c' contains a non-trivial reference to c. That is, c' is not just an alias to c (which means that c' does not contain "$= c$"), but contains a formula that references c. Cells that do not contain formulas can either be labels (if they are not used by other cells) or input cells (if they are used by formulas within other cells). Cells that contain formulas can either be intermediate cells (if they are used by formulas within other cells) or output cells (if they are not used by formulas in any other cells).

Neighbors: While creating spreadsheets, users tend to arrange cells within the grid structure such that spatially co-located cells are usually related. This observation has been supported by our inspection of the EUSES spreadsheet corpus [52]. We call these cells *spatial neighbors* and consider them members of a table or block within a spreadsheet. We have also observed that, in some cases, cells are similar with respect to the underlying model of the spreadsheet program. We refer to these cells as *conceptual neighbors*. Conceptual neighbors might not be spatially co-located.

Chapter 3 – Literature Review

The widespread occurrence of errors in spreadsheets has spurred research targeted at detecting or even preventing them. The distinction between different types of spreadsheet errors is important since they require different types of error detection and prevention techniques. There have been different classifications of spreadsheet errors [92, 97, 104, 115]. Panko and Halverson classified quantitative errors in terms of three main types [92].

1. Mechanical errors: These errors are simple slips that may arise due to carelessness, mental overload, or distractions. Examples include mistyping a number or a reference, pointing at a wrong cell address, or selecting an incorrect range of values or cells.
2. Omission errors: These errors arise from the programmer leaving something out of the model by accident.
3. Logic errors: These errors are caused when the programmer chooses an incorrect algorithm to solve the problem. Such errors typically manifest themselves as incorrect formulas. These are the hardest to detect since they oftentimes require domain-specific knowledge.

Studies have shown that when end users are faced with failures in their spreadsheets and are not sure of the underlying model, they overwrite the cell formulas with a constant

value (the expected output for the cell) [34]. This approach might solve the problem they are facing at that point (an error with an unknown cause in a formula cell), but in the long run, it degrades the maintainability of the spreadsheet and generally introduces errors.

Teo and Tan have classified errors in spreadsheets as either *quantitative* errors or *qualitative* errors. Quantitative errors usually manifest themselves as incorrect results in the spreadsheet. Qualitative errors, on the other hand, take the form of poor spreadsheet design and format. Qualitative errors in a spreadsheet might not be visible right away, but they lead to quantitative errors in future spreadsheet instances [115, 116]. Researchers have studied the occurrence of qualitative and quantitative errors and have made recommendations on how to avoid them. Mason and Keane have suggested that organizations should have a "Model Administrator" (along the lines of Database Administrators for enterprise-level databases) to regulate spreadsheet models within the organization [78]. Williams has recommended the adoption of organization standards [123]. These standards would consist of recommendations for best practices that cover spreadsheet specification, documentation, maintenance, and security that employees should follow while creating spreadsheets. The use of automated spreadsheet audit software has been recommended by Simkin [110]. Teo and Tan have shown that spreadsheet errors are hard to detect during "what-if" analyses if the spreadsheets are not well designed [116]. Based on their studies, they have recommended that courses designed for teaching spreadsheets should focus more on spreadsheet design, targeting ease of maintenance and debugging, rather than demonstrating the features available in spreadsheet systems.

Panko has shown that spreadsheet auditing, as an error-detection technique, takes too much time and effort and requires a lot of expertise [89]. Code inspection of spreadsheet formulas done by individuals working alone has been shown to detect 63% of errors, and group code inspections has up to 83% success rate at detecting errors [90]. Moreover, the costs involved are already very high for medium-size spreadsheets [90] and get worse for large spreadsheets that are used in business [34, 38]. Traditional spreadsheet auditing techniques (without tool support) do not scale very well because the visible area is limited by the size of the screen and the inspections are done at the cell level.

The main problems related to spreadsheets arise from implicit logic, inaccessible model structure, data dependency, and lack of a unifying model base [19, 67]. There are multiple views that can be taken of a spreadsheet: as a program, file, model, and data, all at the same time. Isakowitz et al. recommend the treatment of spreadsheets from two different perspectives: logical and physical. The logical level deals with the model-related and formal aspects, and the physical level concerns details such as storage, formatting, user interface, and other aspects related to the implementation of the model. Additional information about the underlying model would be helpful to the programmer in understanding the spreadsheet and can be obtained through the following activities [105].

1. Detecting homogenous areas in the spreadsheet (a few examples of *similarity conditions* that can be exploited to detect homogenous regions have been described in Section 2.3).
2. Analyzing the data flow in order to find regularities between different spreadsheet regions.

3. Finding logical relationships between cells that are not physically adjacent.

Field audits have shown that spreadsheet programs in business applications can become very large over a period of time. These spreadsheets usually consist of a set of unique formulas that are copied throughout the spreadsheet and then often only slightly modified, usually to change constants or relative references [34]. Some of the spreadsheets developed in the scientific computing domain, for example, use logically equivalent formulas to simulate iteration [49]. Identification of such logically related blocks would help with comprehension and testing of spreadsheet programs.

Visualization techniques have been developed to help end users understand their spreadsheets better [20]. Such visualizations help users identify regions within their spreadsheets, thereby making it easier to identify cells that are outliers [66, 105].

Research on spreadsheets has been aimed at achieving *correctness* or *consistency* of spreadsheet formulas. Correctness of a spreadsheet (as with any other program) is always judged with respect to a specification. Therefore, to be able to say that a spreadsheet is correct, we need information about the specification the spreadsheet was developed for. The spreadsheet generator Gencel [43, 44] uses spreadsheet specifications expressed as templates in ViTSL (Visual Template Specification Language) [12] to generate spreadsheets that are provably free from reference, type, and range errors. Visual languages that allow the user the provide more information about the underlying spreadsheet model have been described in [12, 42]. Once again, the motivation behind these approaches is to be able to guarantee correctness properties of the generated spreadsheet with respect to the specification. Testing is also aimed at ensuring the spreadsheet meets its specifications [100]. While testing cannot guarantee correctness, thorough testing can

at least make the claim that the program works well enough for the most part. Assertions have also been used to express the expectations for values in cells [29]. Assertions on the expected values in input cells can be propagated to the output cells and the system can report violations of any assertion.

To be able to check for correctness, the specification for a spreadsheet has to be provided by the user in formal notation in the case of the approaches in [42, 44], as test cases in the approach in [100], as assertions in [29]. These approaches require extra effort from the user in addition to the actual creation of the spreadsheet. Another problem is that, in many cases, the specification might not be clear when the user starts with the task of creating the spreadsheet. In such situations, the development of the spreadsheet might be more exploratory in nature and the user might not be willing to invest the effort at the end of the task to put together the specification. For example, even with the many benefits offered by the formal approaches mentioned above, not too many organizations might be willing to invest the effort to develop specifications for legacy spreadsheets.[1]

Consistency checking, on the other hand, does not require the spreadsheet specifications as input. In the approaches described in [15, 18, 35] the consistency checkers use extra annotations provided by the user to judge if the formulas in the spreadsheets violate any of the formal rules specified in the respective systems. In the approach described in [1] the system automatically infers the headers for the data cells within the spreadsheet and then uses the header information to see if the formulas in the spreadsheet violate any of the rules of the underlying formal system [45].

[1] This problem has motivated our work on inferring templates automatically [5].

One advantage offered by (some of the) consistency checkers is that they require little [15,18,35] or no [11] extra effort from the user. Coupled with good error messages and explanation systems, the consistency checkers might also be easier for end users to work with. However, the ignorance of the specification is also an inherent problem with consistency checkers that are available today. Because of this problem we might encounter a scenario in which a highly consistent spreadsheet (one in which the formulas pass consistency checks) is also incorrect (does not meet the specification).

In the remainder of this chapter we describe some of the systems that are related to the approaches described in later chapters. In the next section, we describe some of the spreadsheets systems that have been developed for commercial or research use. In Section 3.2, we discuss approaches to spreadsheet correctness that rely on an underlying model. Static analysis tools that carry out consistency checking of spreadsheet formulas are described in Section 3.3. Frameworks for testing of spreadsheets and automatic test-case generation support are described in Section 3.4. In Section 3.5, we describe an approach aimed at helping end users debug their programs because it is related to the spreadsheet debugger GoalDebug described in Chapter 7.

3.1 Spreadsheet Systems

Among the commercially available spreadsheet systems, Microsoft Excel is the most widely used [79]. The other commonly used spreadsheet systems are Gnumeric from Gnome [57] and Calc from OpenOffice [86]. Google's spreadsheet engine is a recent addition to the list of widely available spreadsheet systems [60]. In spite of the large vol-

ume of research that has shown the prevalence of errors in end-user spreadsheets, none of these systems are geared towards detection or prevention of errors. They only have the most rudimentary support for spreadsheet auditing. For example, Excel has an option in which the user can view the data flow within a spreadsheet. Instead, these systems are geared more towards richer user interaction mechanisms, better tools for generating visualizations of spreadsheet data, support for more built-in functions, greater speed, and support for collaboration between users. Except for Microsoft Excel, the other spreadsheet systems can be used free of charge. This aspect enables a much larger population of end users to create their own (potentially error-prone) spreadsheets.

The Forms/3 system has been developed as a research vehicle for the spreadsheet programming paradigm [28]. Unlike the commercial spreadsheet systems mentioned above, spreadsheets in Forms/3 do not have a grid structure or a global addressing scheme based on it. Instead, cells in Forms/3 spreadsheets can be arranged any way the user wants to. The cells can be named by the user (or default to a system-generated name) and the cell values are accessed using the cell name. Forms/3 has the following advanced features not found in the currently available commercial spreadsheet systems.

1. Users can display formulas contained within cells. More importantly, users can also view data flow between definitions and uses of cells.
2. The WYSIWYT testing framework has been integrated into the system and allows the user to carry out systematic testing of their spreadsheets. WYSIWYT and its associated testedness and fault-localization feedback systems will be described in detail in Section 3.4.1.
3. Forms/3 also has an automatic test-generation mechanism, called “Help Me Test”

(HMT), which will be described in Section 3.4.2.

4. Current research on the system is directed towards building a robust explanation system, which is an important component of the "Surprise-Explain-Reward" (SER) strategy [95]. This approach seeks to excite the curiosity of the users towards new features, provide good explanations about features, and show the users the tangible benefits when they use new features. The goal is to use this strategy to lead the users to adopt more and more of the advanced features offered by the system.

Forms/3 has been used very successfully as a vehicle to carry out empirical studies involving the features mentioned above [50, 94, 95, 103].

A few other spreadsheet systems and systems that exploit the spreadsheet paradigm as an interface to other programming languages are briefly described below:

1. Haxcel [77] is a spreadsheet-like interface to the functional language Haskell [93].
2. ModelMaster is a spreadsheet implementation in Prolog that supports structure discovery and querying using Prolog predicates [87]. The system has a frontend implemented in Java.
3. CoreCalc is a spreadsheet implementation in C# [108].
4. The spreadsheet system Intellisheet extends the spreadsheet paradigm and allows the entry of expressions that represent linear and finite-domain constraints, along with arithmetic expressions, in individual cells [13]. The system has constraint solvers that solve the constraints entered in the cells and the system displays the resulting values.

5. The spreadsheet system described in [113] also allows the user to specify constraints within cells. Recomputation of the spreadsheet (possible after a change has been made) in this context involves checking if all the constraints in the sheet have been satisfied, and finding concrete values for cells that satisfy the constraints.

3.2 Specification-Based Approaches

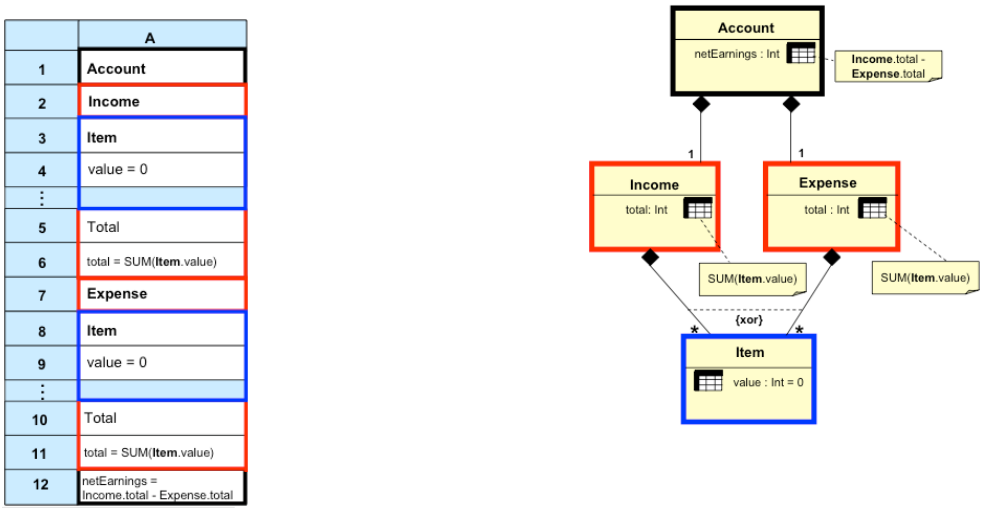
Studies on spreadsheet errors have found that 65% of all spreadsheet errors are contained in formulas [25]. Existing spreadsheet systems impose a model of a flat collection of cells that do not contain any structure other than the arrangement of cells within a grid. While this simple model makes it easy for a user to get started with programming spreadsheets, the lack of modularity and abstractions makes more complicated models increasingly difficult to program and maintain. Another problem is that cells are identified by global row and columns numbers (letters) so that references to cells or subareas of a spreadsheet have to be expressed using these global addresses. The global addressing scheme has been blamed for complicating the comprehension of spreadsheets and for location errors [76]. The rigid, global addressing scheme makes computations vulnerable to changes in the structure of the spreadsheet—much like with assembly language programming where the introductions of a new item into the memory could cause some references to become invalid. Related is the problem of "viscosity", which refers to the difficulty of changing one part of the program without changing other parts [23].

Depending on the application, a spreadsheet can be viewed from different perspec-

tives. The *grid* model of a spreadsheet consists of values in cells that are part of a grid and is a very low-level view of the spreadsheet. The *program* model of a spreadsheet views it as a collection of input and output cells. This perspective takes into consideration the data dependencies between the cells and not their relative spatial location within the grid. The *conceptual* model of a spreadsheet deals with the user's perception of how the spreadsheet works. This view gives some indication of what the user was trying to do when he/she created the spreadsheet. Knowledge about the conceptual model facilitates judgments whether or not modifications to the spreadsheet are correct.

To the best of our knowledge, except for the Visual Template Specification Language (ViTSL) described in Section 4.1 and ClassSheets described later in this section, there are no systems or modeling languages that support end-user design leading to a formal specification. Specifically, in end-user domains there is no support for the capture or maintenance of the formal specifications as a software artifact, and most of the end-user programming environments are more geared towards the exploratory development of programs.

ClassSheets is a visual language for the creation of spreadsheet specifications [42]. The language is built on ViTSL, but offers more sophisticated features and is more related to object-oriented design languages. A ClassSheets specification for simple accounting spreadsheets is shown in Figure 3.1(a). The class names are set in boldface (for example, **Account** and **Income**) in contrast to attribute names and labels which are set in normal face (for example total and value). Therefore, a ClassSheet consists of a list of attribute definitions grouped by classes and arranged within a two-dimensional grid. The grid layout has been adopted to preserve closeness of mapping with spread-



(a) (b)

Figure 3.1: ClassSheets specification (a) and the corresponding UML class diagram (b) for accounting spreadsheets

sheets and also to specify the spatial evolution of the generated spreadsheets. In the grid layout, colored borders are used to separate different classes within the ClassSheet specification. In the specification shown, we have an enclosing aggregating class **Account** with two enclosed aggregating classes **Income** and **Expense**. Both, **Income** and **Expense** can have **Item** instances. This nesting reflects the fact that the generated spreadsheet is likely to have multiple items under income and expenditure. The formulas show how the attributes total (under **Income** and **Expense**) and netEarnings (under **Account**) are computed. The design structure is directly reflected in the UML diagram shown in Figure 3.1(b). A ClassSheet specification an be translated into the

corresponding ViTSL specification.

3.3 Static Checking

Empirical studies have demonstrated the defect-detection capabilities of static type checking [55, 96]. Even though the benefits of type systems are widely accepted, not many studies have been carried out to test or compare the usability of type systems in general-purpose programming languages. Usability issues could also be the reason static type checking has not been incorporated into programming environments targeted at end users. Many researchers have looked at errors in programs developed by novice programmers [41, 111, 112]. However, the frequency and impact of type errors in novice programs is not known.

Analyses using dimensions (mass, length, time, etc.) for prevention of errors in programs was proposed in [69]. Such approaches provide a more fine-grained context for consistency checking than plain types. Physical units (meters, grams, seconds, etc.) have been used to detect unit errors in C programs [68]. As part of the DARPA program for High Productivity Computing Systems, Sun Microsystems is developing Fortress, a language targeted at large-scale scientific computing [53]. Fortress provides support for static checking of units and physical dimensions, and allows programmers to define new dimensions and units and to write programs that are polymorphic with respect to specific dimensions and units [16].

The approach originally proposed in [45] uses headers used by end users to label the data in their spreadsheets as implicit unit declarations, and carries out consistency

checking based on the allowed combinations of units. For any cell, the user would have to specify the labels that can be used to infer the units. As a follow-up, the authors proposed a mechanism by which the users could customize the inference mechanism in [30]. In order to minimize the effort required of the user, we developed a set of heuristics that automatically infer the headers for the cells within a spreadsheet [1]. The automatic header inference is employed by the UCheck system [11] to carry out fully automatic consistency checking using the formal rules defined in [45]. UCheck is described in detail in Chapter 5. In this section, we describe a few of the systems that have incorporated some of these ideas into consistency checkers for spreadsheets.

3.3.1 Unit Checking

The system described in [15] extends the approach from [45] by distinguishing between two kinds of relationships between cells and their headers. The *is-a* relationship indicates instances and subcategories and the *has-a* relationship describes properties of the data items or sets. We will explain how the system works by means of an example. For details about the inference rules, please refer to [15].

Cell B4 in Figure 3.2 has two headers: TVs and 2001. Furthermore, TVs *is-a* (subcategory of) Electronics and 2001 *is-a* (instance of) Year. We can also say that the set of TVs *has-a* (property called) Total. The hierarchical *is-a* unit structure is denoted with square brackets. Since the data in a cell can be uniquely identified by one *has-a* relationship, the system allows a single *has-a* component for the unit of a data cell. The *has-a* component is denoted by braces. The inferred unit for cell B4 has two components that

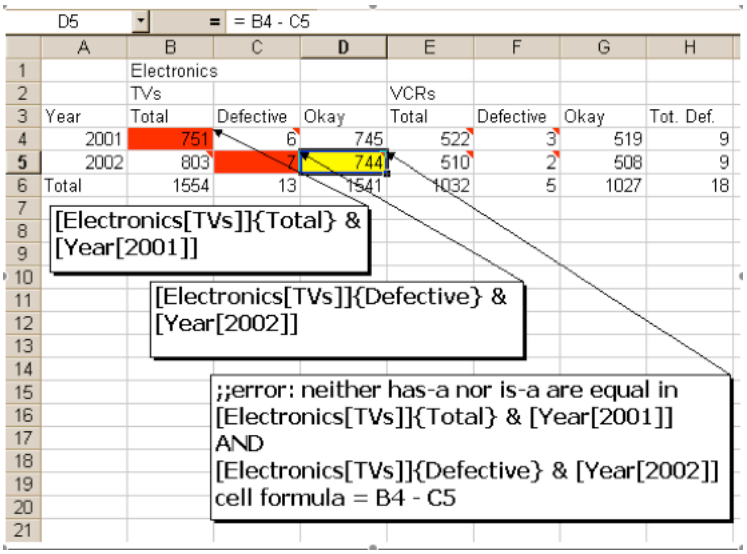


Figure 3.2: Unit checking

are combined using the & operator:

$$[\text{Electronics}[\text{TVs}]]\{\text{Total}\}\ \&\ [\text{Year}[2001]]$$

For the spreadsheet in Figure 3.2 the inferred unit for cell C5 is

$$[\text{Electronics}[\text{TVs}]]\{\text{Defective}\}\ \&\ [\text{Year}[2002]]$$

The formula in cell D5 is B4 - C5 and the inferred unit is

$$[\text{Electronics}[\text{TVs}]]\{\text{Total}\}\&[\text{Year}[2001]] + [\text{Electronics}[\text{TVs}]]\{\text{Defective}\}\&[\text{Year}[2002]]$$

This unit is not well formed as per the rules since the two components differ in both the

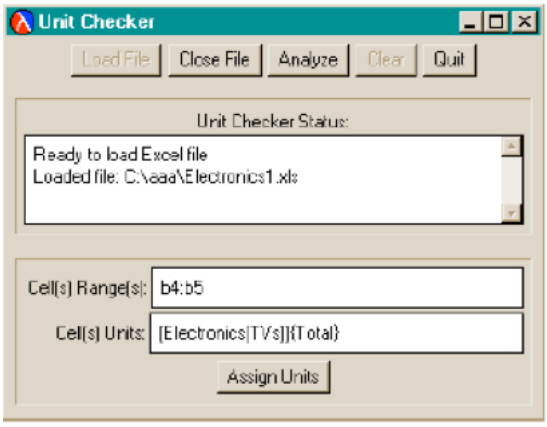


Figure 3.3: Specifying header structure in unit checker

has-a (Total and Defective) part and the *is-a* (2001 and 2002) part.

This system requires the user to specify the header structure and distinguish between the *is-a* and *has-a* relationships between the cells and the headers. The interface is shown in Figure 3.3.

As can be seen from Figure 3.2, the rather cryptic error messages displayed to the user simply describes the error in terms of the underlying rule system and do not suggest what course of action the user might take to correct the error.

The units inferred by the system are ambiguous with regard to the *has-a* component. For example, in the spreadsheet shown in Figure 3.2, the unit inferred for cell B4 (based on the unit annotations provided by the user) is

[Electronics[TVs]]{Total} & [Year[2001]]

The *has-a* component could as well have been associated with the year information to

B4:	[Electronics[TVs]]{Total} & [Year[2001]]
B5:	[Electronics[VCRs]]{Total} & [Year[2001]]
B6=B4+B5:	[Electronics]{Total} & [Year[2001]]
C4:	[Electronics[TVs]]{Total} & [Year[2002]]
C5:	[Electronics[VCRs]]{Total} & [Year[2002]]
C6=C4+C5:	[Electronics]{Total} & [Year[2002]]
C7=B6+C6:	[Electronics]{Total} & [Year]
D4:	[Electronics[CDs]]{Total} & [Year[2001]]
D5:	[Electronics[DVDs]]{Total} & [Year[2001]]
D6=D4+D5:	[Electronics]{Total} & [Year[2001]]
E4:	[Electronics[CDs]]{Total} & [Year[2002]]
E5:	[Electronics[DVDs]]{Total} & [Year[2002]]
E6=E4+E5:	[Electronics]{Total} & [Year[2002]]
E7=D6+E6:	[Electronics]{Total} & [Year]

Figure 3.4: Unit inference example

give the following unit:

[Electronics[TVs]] & [Year[2001]]{Total}

This unit implies that 2001 *has-a* (property called) Total. Even though both the above units are conceptually the same (and correct), the inference rules treat them differently, thereby giving rise to false positives during unit checking.

The system also generalizes units too soon, thereby weakening the unit checking. Consider a spreadsheet that contains the cells and their corresponding units as shown in Figure 3.4. The base units are shown for the cells with constants and inferred units are shown for cells with the addition operation.

We see that the inferred units in C7 and E7 are the same even though the underlying

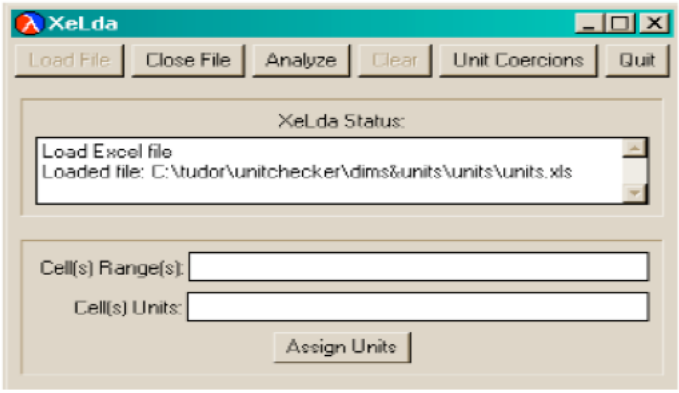


Figure 3.5: Specifying headers and dimensions in XeLda

is-a components that are lost during generalization are different. The system also allows operations like B6+E6 and C6+D6. Both operations should be disallowed because the *is-a* components of both Electronics and Year are different. Moreover, the inferred units for these two operations are the same: [Electronics]{Total} & [Year]—a consequence of the fact that eager generalization removes the information about the underlying unit structure. This problem of eager generalization gives rise to false negatives during unit checking.

3.3.2 XeLda

XeLda is a consistency checker that validates the *unit* correctness of spreadsheet formulas [18]. XeLda works with physical units like kg·m/s^2.[2] XeLda's control panel for loading a spreadsheet for consistency checking is shown in Figure 3.5. The interface can

[2]The standard symbol for product of units is ·, which is what we will use in our description of the system, but XeLda uses - (as can be seen in the screenshot in Figure 3.6).

also be used to specify the unit annotations for cells within the spreadsheet. The unit annotations can also be entered within Excel's comment boxes attached to individual cells. Any cell that has a number needs to be annotated with units, otherwise the system treats it as a dimensionless constant. XeLda computes the units for the cells with formulas. Any units specified by the user for cells with formulas are treated as unit assertions for those cells. XeLda detects the following kinds of errors.

1. A *consistency error* occurs when the formula in a cell does not use units in a consistent manner. Cells that contain consistency errors are shaded yellow.

2. A *match error* occurs when the derived unit for a cell does not match its unit annotation. Cells that contain match errors are shaded orange.

In addition to match and consistency errors, XeLda also marks cells that contain references to cells with errors as the sites of *propagation errors*. Cells with propagation errors are shaded purple.

Examples of unit errors detected by XeLda are shown in Figure 3.6. Cells A2, C2, and B5 in the spreadsheet have been annotated with the units kg, $m \cdot s^2$, and $kg \cdot m/s^2$, respectively. The inferred unit for the formula A2*C2 in cell B5 is $kg \cdot m \cdot s^2$. XeLda detects a match error in this case and shades the cell B5 orange because the inferred unit does not match the unit annotation for the cell.

XeLda flags the formula A9 $<$ C9 in B12 as a consistency error since the unit annotations of cells A9 (apples) and C9 (oranges) do not agree. XeLda would also flag the formula A9+C9 as a consistency error since the units do not agree. In this case, the system is too restrictive since the sum would be 90 fruits, as would be inferred by UCheck

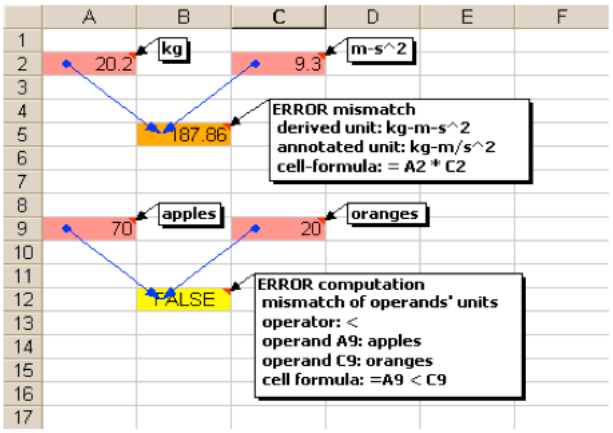


Figure 3.6: Dimension checking in Xelda

and the system described in Section 3.3.1.

Requiring the users to manually annotate the cells with the unit information introduces another potential source of error. User-annotation errors can be minimized if XeLda could automatically infer physical units from header information in spreadsheets.

XeLda supports unit coercions. For example, the user can specify that 1 meters is equal to 3.3 feet and XeLda will automatically carry out conversions when the user combines cells that have meters and feet as their units. The results are presented in the canonical unit chosen for each equivalence class of units. XeLda does not support unit synonyms. For example, Newton and $kg{\cdot}m/s^2$ are considered different units by the system.

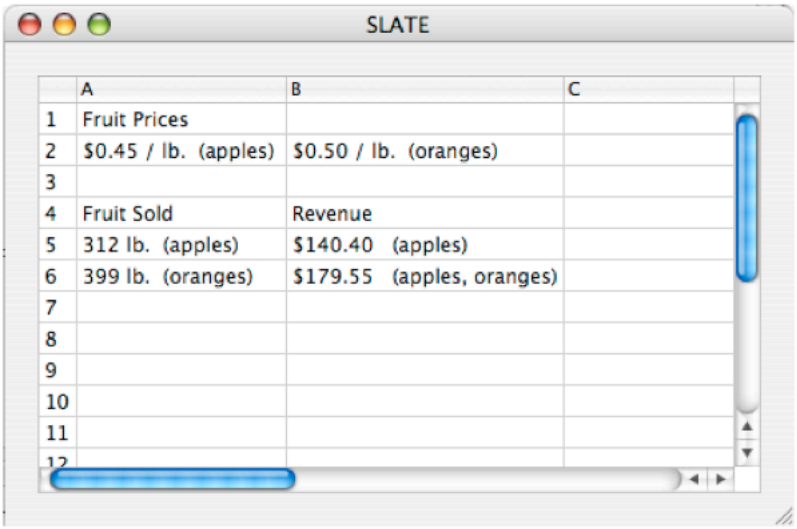


Figure 3.7: Cell annotations in Slate

3.3.3 SLATE

SLATE is a spreadsheet consistency checker that combines information about physical units and labels to detect errors [35]. Every input cell in SLATE has two attributes: a unit and a label. In the spreadsheet shown in Figure 3.7, cell A2 has unit $/lb and label apples and cell B2 has unit $/lb and label oranges. The formula A2*A5 in B5 computes the total revenue from the sale of apples and has the unit $ and label apples. The formula A2*A6 in B6 incorrectly computes the revenue from the sale of oranges by multiplying the weight of oranges sold with the rate for apples. The derived label for B6 reflects this error and has attributes from both apples and oranges.

One common disadvantage of all the systems discussed in this section is that they require the user to annotate the spreadsheet cells with the header or unit information. In addition to the user effort involved, manual annotation of the unit information for the

cells is prone to errors. For example, the user might make a mistake while entering the unit information or fail to specify the unit information for a cell. In such cases, the user would have to invest further effort in debugging the errors in the unit structure.

3.4 Testing

Given the dilemma that testing does not show the absence of faults within a program, when should a practitioner consider a program well tested? Researchers have developed test adequacy criteria which allow the tester to decide when to stop testing. Different test adequacy criteria have different levels of confidence about the absence of faults in the program that is being tested.

For a program p, $p(t)$ denotes the output of the program for the input t. The set of all valid inputs of p is its domain D. If p implements function f on domain D, then the test set $T \subset D$ is *reliable* for p and f if $\forall t \in T, p(t) = f(t) \Rightarrow \forall t \in D, p(t) = f(t)$ [65]. In effect, the successful execution of a reliable test set guarantees that the program satisfies the functional specification on all inputs. However, Howden has shown that even though finite reliable test sets exist, there is no effective procedure for generating reliable test sets of finite size [65]. Therefore, the notion of *test case adequacy* has been proposed as an alternative to measure the effectiveness of test cases. Let Q denote the (infinite) set of faulty programs. A test set $T \subset D$ is said to be *adequate* for p and f if $\forall q \in Q, \exists d \in D, q(d) \neq f(d) \Rightarrow \exists t \in T, q(t) \neq f(t)$ [26, 39, 59]. As is obvious from the definitions, the notion of test adequacy is geared towards detection of faults rather than showing correctness.

Budd and Angluin have studied what it means for test data to demonstrate correctness of programs. They have shown under what conditions such data can be automatically detected and/or generated [26]. Detecting that the test data meet the requirements to show the correctness of the program is done by an *acceptor*, which is important to let the tester know that no further test cases need to be considered. Using a *generator* to generate test cases that meet a criterion helps with effective testing. Unfortunately, as shown in [26], there is no effective procedure for either generating adequate test sets or for detecting that a given test set is adequate.

In the case of *relative adequacy* (also referred to as *mutation adequacy*), the number of incorrect programs considered is greatly reduced by only considering "almost correct" programs. In this case, given the finite set Q' of programs that are "almost correct", a test set $T \subset D$ is adequate for p relative to Q' if $\forall q \in Q', \exists d \in D, q(d) \neq f(d) \Rightarrow \exists t \in T, q(t) \neq f(t)$ [39]. The rationale behind considering only "almost correct" programs in Q' is the *competent programmer hypothesis*. The hypothesis states that although programmers produce buggy code, it works correctly for most inputs and fails only in few cases.

One often employed test adequacy criterion is *definition-use (du)* adequacy, which requires that each possible path from any definition of a variable to all its uses be executed by test cases. Du adequacy is a data-flow-based adequacy criterion. Commonly used control-flow-based test coverage criteria (each one has it corresponding adequacy measure) are the following.

1. *Statement coverage* is the percentage of the lines of code that have been executed during testing.

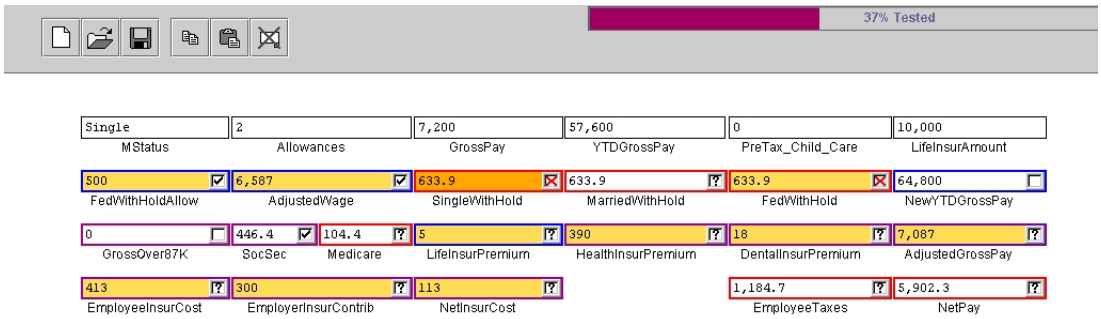


Figure 3.8: Fault localization feedback in WYSIWYT

2. *Branch coverage* is the percentage of the branches within the code that have been executed during testing.

3. *Path coverage* is the percentage of possible paths from entry to termination of the program that have been executed during testing.

The relative effectiveness of du-adequacy and other adequacy criteria at fault detection have been compared in [84, 121] and it has been shown that du-adequacy is a stronger test adequacy criterion than the control-flow-based criteria listed above.

3.4.1 "What You See Is What You Test" (WYSIWYT)

Rothermel et al. have developed the "What You See Is What You Test" (WYSIWYT) methodology for testing spreadsheets [100]. The WYSIWYT methodology has been implemented in the Forms/3 spreadsheet engine. A Forms/3 spreadsheet is shown in Figure 3.8.

WYSIWYT uses data-flow-adequacy criteria and coverage monitoring to give users

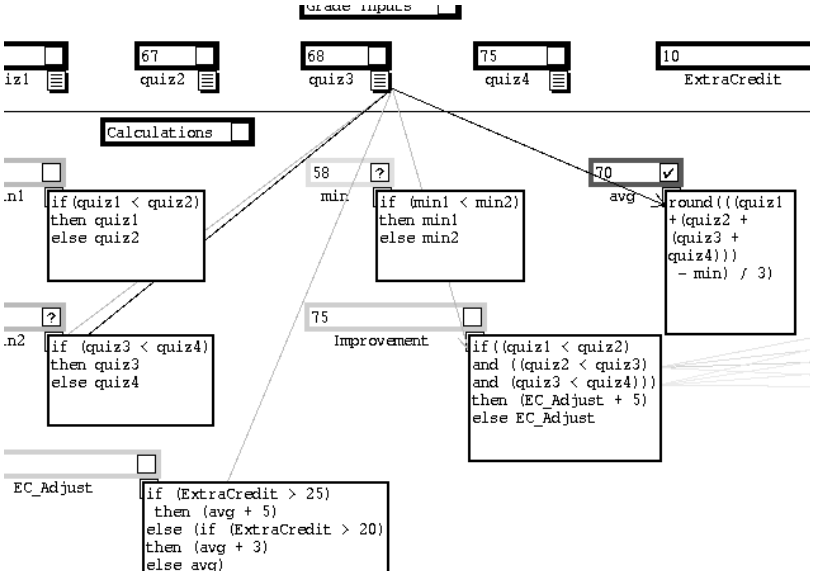


Figure 3.9: Arrows showing definitions and uses in Forms/3

incremental feedback (using cell border colors) about the percentage testedness of the formulas within the spreadsheet cells. Cell border colors can range from red (denoting the fact that the cell is untested), all the way to blue (denoting the fact that the cell formula has been completely tested). Input cells do not have any border shading. The overall testedness of the entire spreadsheet is indicated by a progress bar at the top. For example, 37% of the du associations in the spreadsheet shown in Figure 3.8 have been exercised using test cases. The system also allows users to expand formulas within the cells and view the data flow as shown in Figure 3.9. The arrows connect definitions and their use within formulas (and subformulas), and the shading of the arrows indicates if the du pair has been tested or not.

For a given set of input values, the user can inspect the output in a cell and mark

it (in the top right corner) with a ✓ if the output is correct. On the other hand, if the output is incorrect, the user can mark the cell with a ✗ to indicate an observed failure. The system keeps track of the ✗ marks placed by the user and employs dynamic slicing techniques to compute the fault likelihood of the cells that are upstream (in the data-flow chain) from the marked cell. The fault likelihood feedback is presented to the user in terms of cell background shading—a white background indicates that the cell formula is probably correct, while dark orange background indicates that it is highly likely that the formula in the cell has a fault.

The Forms/3 spreadsheet system has been the vehicle for numerous user studies. Studies have shown that 81% of seeded faults were detected by participants using the implementation of WYSIWYT within the Forms/3 spreadsheet system [100]. Studies have also shown the effectiveness of the fault-localization mechanism of Forms/3 [95, 101]. Empirical evaluations have also been carried out to explore different ways to map the error information to visual feedback in Forms/3 by itself [103], and when combining the fault-detection capabilities of multiple systems [73, 74].

3.4.2 "Help Me Test" (HMT)

Testing can expose faults within programs. However, it is not reasonable to expect end users to create effective test cases to test complicated spreadsheets. This problem has motivated research into automatic test-case generation systems. The "Help Me Test" (HMT) system that has been integrated into Forms/3 and the WYSIWYT testing framework generates test cases to support incremental testing targeting du adequacy [50, 51].

HMT basically uses two algorithms for test-case generation. *Random* generates test cases by randomly selecting input values [21]. *Chaining* first build a *constraint path* between a definition *d* and an use *u*. A constraint path is basically the sequence of nodes and branches that starts at *d* and ends at *u*. In the second step, the chaining algorithm compares the constraint path to the execution traces of the formula that is being tested. It determines the changes that have to be made to the inputs so as to execute the constraint path and reports the new set of inputs as the new test case [48].

The test cases generated by HMT using the above algorithms can be refined with additional user input in the form of *assertions* on cell values. Assertions in Forms/3 specify the allowed range of values for a cell [29] and help with automatic test-case generation by limiting the range of input values that need to be considered.

3.5 Debugging

Whyline is a debugger that has been prototyped in the Alice programming environment [72]. The system allows the programmer to chose "Why did" and "Why didn't" questions about their program's behavior. The questions are automatically generated based on static and dynamic analyses of the programs. Whyline provides answers to the programmer's questions in terms of the runtime events that caused (or didn't cause) the desired (or unexpected) behavior.

In the scenario shown in Figure 3.10, the programmer is debugging a Pac-Man game implemented in Alice. Contrary to the programmer's expectations, in the implementation, Pac does not disappear when it comes in contact with a ghost. The system allows

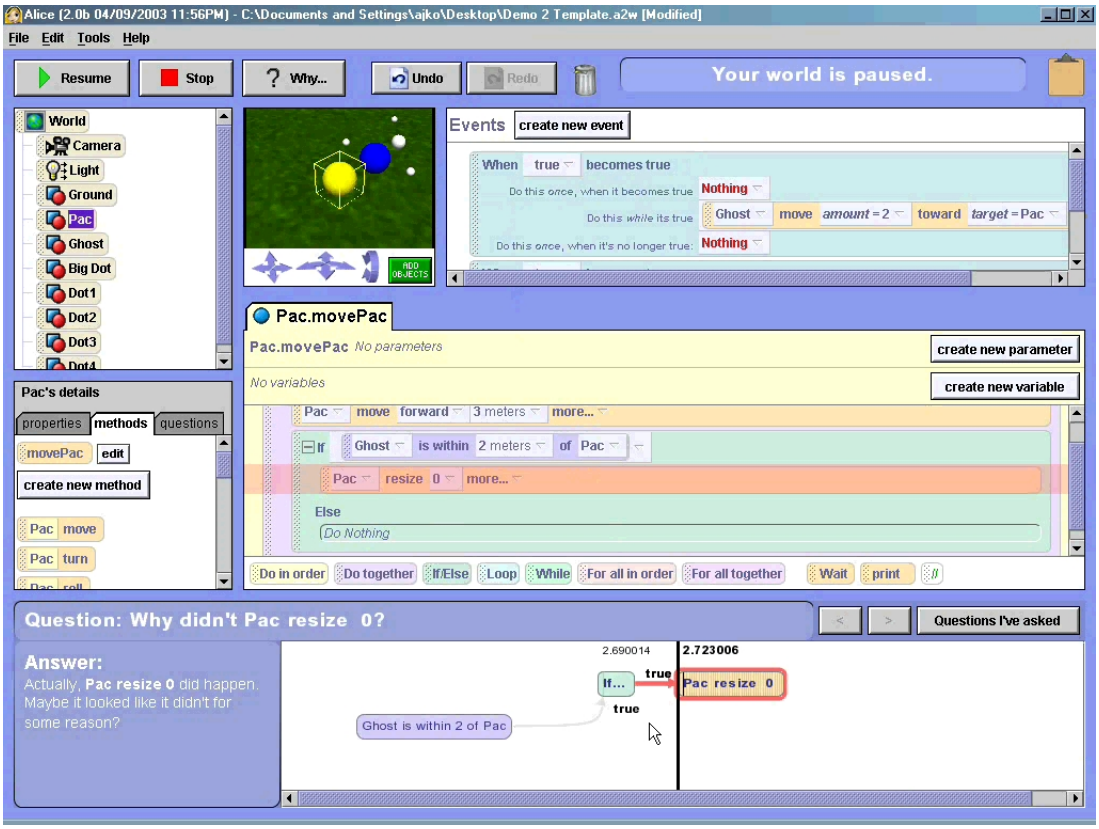


Figure 3.10: Debugging using Whyline

the user to pick the question “Why didn’t Pac resize 0?” and explore the computations that led to Pac not disappearing.

In user studies, it has been shown that programmers using Whyline debugged Alice programs up to 8 times faster than programmers who debugged without Whyline. A similar approach has been developed for debugging constraints [124]. Both, Whyline and the system described in [124], are similar to GoalDebug (described in Chapter 7) in that they exploit the user’s expectation about the behavior of the program to locate faults within programs. GoalDebug goes one step further and generates a list of change

suggestions, any one of which, when applied, would result in the expected program output.

Chapter 4 – Specifications-Based Approach to Spreadsheet Correctness[1]

> *Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so.*
>
> Douglas Adams

All the updates performed on a spreadsheet can be classified under one of two categories. Formula edits modify the spreadsheet program logic, whereas changes on data cells modify the inputs to the spreadsheet program component. In the traditional approach to spreadsheet development that is supported by most commercially available spreadsheet engines, the program and data/input updates are applied to a common representation. Moreover, the effects of the changes are immediately visible, whereas the changes themselves are not. For example, to view all the formulas in a spreadsheet in Microsoft Excel, the user has to explicitly invoke the formula audit mode. From a broader perspective, we have the following kinds of spreadsheet updates that can be considered.

1. *Changes to input cells.* In the case of spreadsheets, such changes should not be considered as changes to the program itself. In the case of programs developed in general-purpose programming languages, they are equivalent to rerunning the program on different inputs.

[1] The contents of this chapter are based on [5, 7, 12, 43, 44].

2. *Modifications to the existing formula cells.* These changes would qualify as changes to the spreadsheet program, and might be required due to changes in the requirements or if the user is carrying out exploratory analyses and development.
3. *Addition (deletion) of formulas to (from) the spreadsheet.* Such changes can be considered as extensions to the spreadsheet program. Oftentimes, the addition of new formula cells is accompanied by the addition of corresponding labels, input and intermediate cells as well.
4. *Simulating iteration.* In spreadsheets, users simulate iteration by copy-pasting (or clicking and dragging) cells or blocks of cells. In such cases, addition of one or more such blocks would be similar to increasing the number of iterations during the execution of a program, rather than changing the program itself, in the case of a program written in a general-purpose programming language.
5. *Modifying data records.* Users store related information as blocks of cells within the spreadsheet. In such cases, the blocks represent *data records* (when viewed from a database perspective) and the addition or deletion of such blocks represent the addition or deletion of the corresponding records. In this case, the blocks of cells do not contain any formulas.

Users carry out update operations on their spreadsheet to meet some high-level goal. In the absence of support for higher-level abstractions, users are forced to break up simple high-level cognitive tasks into multiple low-level spreadsheet tasks while mapping their mental models to the spreadsheet grid structure. In such cases, it is easy to introduce errors in the spreadsheet if the user forgets one or more of the steps and leaves the spreadsheet in an inconsistent state. The errors depend on the kinds of updates the user

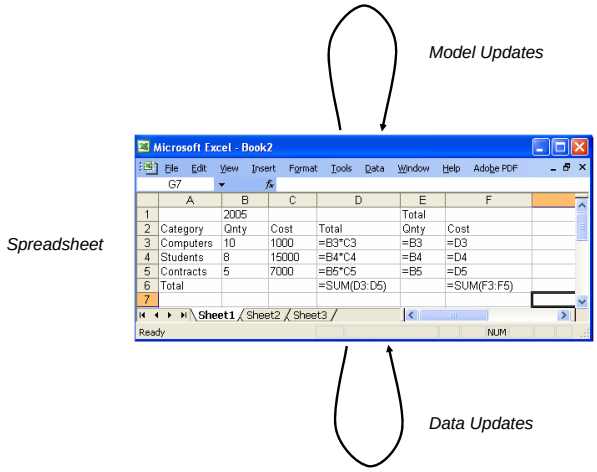


Figure 4.1: "Traditional" approach to spreadsheet development.

was trying to perform.

Why is it important to develop a formalism that allows users to create spreadsheet specifications? Since spreadsheets are programs developed by the user to satisfy some requirement, it is principally impossible to check the correctness of a given spreadsheet without a formal definition of its specifications. Availability of the formal specifications affords the following advantages:

1. The higher-level abstractions would allow the user to record how the spreadsheet is supposed to behave.
2. Formal specifications can be used to check and ensure that subsequent modifications to the spreadsheet program do not violate the specifications.
3. Program generation techniques can be exploited to prevent errors and ensure spec-

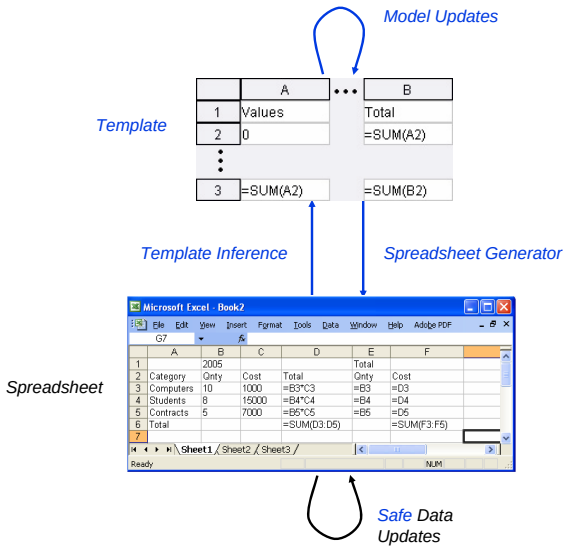


Figure 4.2: Proposed approach to spreadsheet development.

ification correctness once the specification is defined.

The main problem with the current approaches to spreadsheet development is that they do not differentiate between the changes to the data within the spreadsheet and updates to the underlying spreadsheet model. As shown in Figure 4.1, in the "traditional" approach to spreadsheet development, the updates to the data and the model are performed on the same spreadsheet, and there are no restrictions on the kinds of updates that can be performed. The new approach we propose for spreadsheet development separates the model updates from the data updates as shown in Figure 4.2.

In the approach described in this chapter, the spreadsheet model is captured in the form of a template developed in the Visual Template Specification Language (ViTSL) using the ViTSL editor. The template can then be imported into a system called Gencel

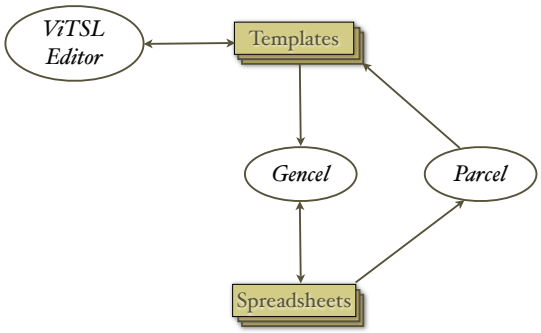


Figure 4.3: ViTSL/Gencel approach to spreadsheet development.

(which is available as an add-in for Excel) and is used to generate spreadsheets that are guaranteed to conform to the model. Legacy spreadsheets can be imported into the ViTSL/Gencel framework using a template parser called Parcel. This approach is shown in Figure 4.3. All model-level changes are performed at the template level and only data updates are allowed within the spreadsheets generated using Gencel.

In the next section we describe ViTSL and an editor that can be used to develop ViTSL templates. In Section 4.2, we describe how Gencel generates spreadsheets based on ViTSL templates. We realize that for wide adoption of the ViTSL/Gencel framework, users would need tool support to infer ViTSL templates from their legacy spreadsheets. To provide this support, we have developed algorithms to automatically infer ViTSL templates from arbitrary spreadsheets. In Section 4.3, we describe these algorithms that have been implemented in Parcel. Since the mapping between templates and spreadsheets is many to many, automatic template inference is highly ambiguous. Therefore, we have conducted a study (described in Section 4.4) to evaluate the quality of templates inferred by Parcel. Since templates capture the model behind a spreadsheet, template

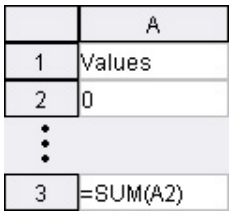


Figure 4.4: ViTSL template for a summation column

inference can be viewed as a way to "compress" a potentially large spreadsheet to its essence. If the inferred template differs drastically from the model intended by the user, it might be indicative of errors or inconsistencies within the spreadsheet. In Section 4.5 we describe how the use of different criteria for template inference can help with the detection of errors within the spreadsheet. We present conclusions and directions for future research in Section 4.6.

4.1 A Visual Specification Language for Spreadsheet Templates

ViTSL allows users to express spreadsheet models as templates. The design of ViTSL [12] has been informed by the Cognitive Dimensions of Notations [23]. ViTSL was designed such that the templates resemble spreadsheets to preserve closeness of mapping between the template specification language and spreadsheets. Consider, for example, the ViTSL template shown in Figure 4.4. As is the case with spreadsheets, the columns are named A, B, and so on and the rows are numbers 1, 2, and so on in a ViTSL template. This template can be used to generate a spreadsheet that has a column of numbers with a label at the top and a summation formula at the bottom. The template has three

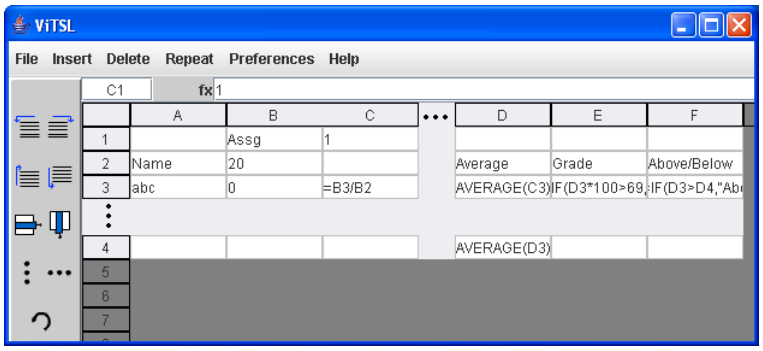


Figure 4.5: Grade sheet template in ViTSL.

components:

1. The column header Values in cell A1.
2. The vertically expanding group (also called *vex group*) in cell A2. The vex group is marked by the ellipsis ⋮ under row 2 and indicates that the row can repeat downwards.
3. The summation formula in cell A3. Since the summation formula has a reference to a cell in the vex group, in the generated spreadsheets, the formula will aggregate over all the cells generated from the vex group.

The template describes a class of spreadsheets that all consist of one column with a header at the top, one or more cells containing numbers, and a cell with a formula computing the sum of all the cells with numbers at the bottom.

A ViTSL template that can be used to generate grade spreadsheets is shown in Figure 4.5. As in the case of the template for the summation column, in the grade spreadsheet template, the ellipsis under row 3 indicate that the row can repeat downwards. In the instances of the grade spreadsheet, the rows store information about individual stu-

	Insert column to the left of current position
	Insert column to the right of current position
	Insert row above current position
	Insert row below current position
	Delete row
	Delete column
	Convert selected block to a vex group
	Convert selected block to a hex group
	Undo

Table 4.1: Toolbar options for constructing ViTSL templates

dents enrolled in the course. The absence of separators between the columns B and C, together with the horizontal dots $\cdots$ to the right indicate that the two columns form a block and can have multiple instances to the right. In instances of the grade spreadsheet, these blocks store information for different assignments in the course. Such columns, or groups of columns that can repeat horizontally are called *horizontally-repeating (or hex) groups*. The formulas that are specified in the ViTSL templates are automatically updated in the generated Gencel spreadsheets. In addition to the formulas, the template consists of labels, such as Assg and Name, that will generally not be edited by the user in the generated Gencel spreadsheet and the sample values, such as 10, abc, and 0, that will be edited.

$f \in Fml$	$::=$	$v \mid \uparrow a \mid \psi(f,\ldots,f)$	(*formulas*)
$b \in Block$	$::=$	$f \mid b \mid b \mid b \hat{} b$	(*blocks, tables*)
$c \in Col$	$::=$	$b \mid b^{\downarrow} \mid c \hat{} c$	(*columns*)
$t \in Template$	$::=$	$c \mid c^{\rightarrow} \mid t \mid t$	(*templates*)

Figure 4.6: Templates.

We have implemented an editor for ViTSL templates in Java. The interface of this editor is shown in Figure 4.5. A user can create a ViTSL template using the options available on the toolbar on the left. The buttons and their corresponding actions are described in Table 4.1. Once the user has created the ViTSL template, it can be saved to a file, which can then be imported into Gencel.

As shown in the examples described above, ViTSL offers the following visual elements for the creation of templates.

1. *Cells*, represented by rectangles and containing formulas.
2. *References*, represented by (names for) relative grid offsets.
3. *Vex groups*, represented by vertical dots that indicate the possible expansion of one or more cells in the vertical direction.
4. *Hex groups*, represented by horizontal dots that indicate the possible expansion of one or more columns in the horizontal direction.

These syntactic elements of ViTSL are summarized in the grammar shown in Figure 4.6. The constructs correspond directly to the visual notation. A *template t* is given by a horizontal composition ($|$) of fixed (c) or horizontally expandable ($c^{\rightarrow}$) columns, where a column is given by a vertical composition ($\hat{}$) of fixed (b) or vertically expandable ($b^{\downarrow}$) blocks. A block is given by a composition of formulas (f). Blocks are also used

to represent plain *tables*. Formulas consist of basic values (v), references ($\uparrow a$), and expressions that are built by applying functions to a varying number of arguments given by formulas ($\psi(f,\ldots,f)$). See also definitions in Section 2.1.

References are given by pairs of integers and represent relative references in the form of offsets. We use the following abbreviations for cell offsets: $\ell = (-1,0)$, $r = (1,0)$, $u = (0,-1)$, and $d = (0,1)$.

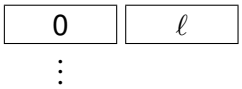
As an example, consider the template for a summation column shown in Figure 4.4. This column is represented by the following template.

$$\mathsf{Values\hat{}0^{\downarrow}\hat{}SUM}(u)$$

Vex and hex groups cannot be arbitrarily nested in ViTSL templates. A condition imposed by the grammar is that hex groups may contain vex groups. In addition, a couple of structural constraints are needed to ensure that a reasonable definition for the spreadsheet update operations exists. For example, all columns in a template have to *align* vertically. To explain the idea of alignment, consider a column as a sequence of fixed and expandable blocks (cell groups), say $c = [b_1,\ldots,b_k]$. Now c matches another column $c' = [b'_1,\ldots,b'_k]$ only if (a) b_i has the same height as b'_i and (b) b_i is an expandable group iff b'_i is. This constraint ensures that vex groups are horizontally aligned, which allows the insert-row command to be defined to insert a number of rows according to the common height of the vex groups. Similarly, we require that all blocks in a column have the same width. For columns in hex groups, this constraint ensures that the insert-column command can be defined to create a number of columns according to the

common width of the blocks of the hex group.

We can consider an example that violates these constraints to see why they are required. Consider the case for the following template:



In this template, we have the horizontal composition of two columns—the left column repeats vertically whereas the right column is simply a reference. These columns do not align and thus the above template is not legal. In the initial table, we would have one value with a single reference to that value, which causes no problems. However, if we consider the insertion of rows, we can observe that additional values would be added. What would be the meaning of the reference in that case? Would it refer merely to the topmost value or to all of the values? In addition, the vertical and horizontal concatenation of blocks assumes that, at all times, blocks, columns and tables will be rectangular. However, we can see a non-rectangular shape emerging in this example. How does such a shape concatenate with other columns or tables? These uncertainties have led us to forbid such templates.

4.2 Generating Correct Spreadsheets using Gencel

Once the user has developed a template in the ViTSL editor, it can be saved to a file and then be loaded into the Gencel system, which has been implemented as an add-on

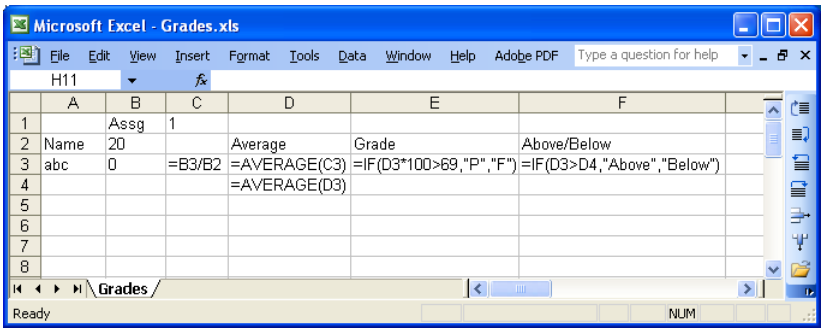


Figure 4.7: Grade sheet in Gencel.

to Excel. The first instance of the grade spreadsheet, created when the grade template is loaded into Gencel, is shown in Figure 4.7. Note the buttons for the Gencel update operations on the toolbar on the right. The two upper buttons allow insertion of a new row above or below the current cursor position. The next two buttons allow the insertion of a column to the left or right of the current cursor position. The two buttons after that are for the deletion of a row or a column, respectively. The last button brings up the file browser interface for loading templates into Gencel.

The key advantage of using Gencel is that spreadsheet updates are customized on the basis of the template that is loaded. That is, structural (insert/delete row/column) and formula updates are carried out by the system such that every instance of the spreadsheet conforms to the ViTSL template that is being used. For example, to insert information about a new assignment, the user of the grade template can simply position the cursor within the region for an assignment and then click one of the buttons for inserting columns (depending on whether the new column needs to be inserted to the left or right of the current position). Gencel automatically detects that the cursor is within a block

for an assignment and performs the insertion of two Excel columns. All formulas are updated automatically and are protected against unintended changes. Along similar lines, to insert information about a new student, the user can position the cursor on a row for a student and then use one of the insert-row buttons. Once again, the system carries out the structural and formula updates automatically. For details about the formal definitions of the functions that carry out the spreadsheet updates, see [44].

Since Gencel performs the updates automatically, the user can concentrate on ensuring the data within the spreadsheet is correct and does not have to worry about updating the formulas. If the ViTSL template matches the specifications, Gencel provably eliminates the following kinds of errors [44]:

1. *Range errors* (for example, omitted or additional cells in aggregation formulas)
2. *Reference errors* (for example, references to wrong cells or circular references)
3. *Type errors* (for example, using strings in numerical computation)

The impact of these errors in real-world spreadsheets have been extensively documented. For example, a range error caused a Florida construction company to underbid a project by a quarter of a million dollars [62]. An example of a type error is the illegal interpretation of a date as a numeric value, which caused an operating fund of the Colorado Student Loan Program to be understated by $36,131 [118]. Finally, a reference error caused a hospital's records to overstate its Medicaid/Medicare crossover log by $38,240 [119]. The use of Gencel would have prevented all these errors.

4.3 Automatic Inference of Spreadsheet Templates

There is a high level of ambiguity in automatic template inference since spreadsheets impose very few restrictions on how the users map their mental models to the two-dimensional grid structure. In general, the mapping between spreadsheets and templates is many-to-many, and we suspect template inference would require some user interaction to resolve ambiguities.

While developing the algorithms for Parcel, we made two design choices about the system functionality.

1. *The generated template should be the smallest one possible, starting from which the user should be able to generate the target spreadsheet using only Gencel insert/delete row/column commands and changes to data cells.*
2. *The system should be tolerant to errors in the spreadsheet. The user should be able to control the error-tolerance threshold.*

How the first choice influences the inference of templates will become clear later in the section. The basic idea is that if template inference generates a set of equally likely templates from a spreadsheet, Parcel reports the one with fewer cells. The second choice aims to accommodate deviations in spreadsheets. These "perceived" deviations could be because of the ambiguity inherent to template inference or because of the presence of errors within the spreadsheet. We hope to be able to exploit this aspect for error detection in later versions of Parcel. For example, when the user picks a particular template as the correct one for the spreadsheet under consideration, the system can report the cells that contain formulas that deviate from the template. These formulas could potentially be

the sites of errors, and need closer inspection since they deviate from the user-approved template.

The structure of templates can be represented by patterns that can be generated from the following grammar.

$$p_x \;\; ::= \;\; (f, x) \mid p_x \cdot p_x \mid p_x^n$$

That is, a pattern p_x can be a formula f paired with some additional information of type x, the composition of two patterns, or n repetitions of the same pattern.

To identify patterns, we need a notion of cell similarity. To this end we employ a predicate $\equiv_\eta$ that is parameterized by the type of similarity used in any given case. The equivalence relationship $\equiv_\eta$ is assumed to be derived from a function $\eta : F \rightarrow x$ that obtains information to be compared from formulas.

$$(a_1, f_1) \equiv_\eta (a_2, f_2) \iff \eta(f_1) = \eta(f_2)$$

The equivalence relationship has been parameterized with the function η because different kinds of templates can be inferred using different equivalence functions. In the case of Parcel, η computes cp similarity as the equivalence function. However, the functions that carry out template inference allow other equivalence functions as well. For example, in Section 4.5 we describe how the use of type equivalence in template inference can be used for error detection.

The equivalence relationship can be extended to columns. That is, we can say that two columns $S\lceil i \rceil$ and $S\lceil j \rceil$ are $\equiv_\eta$-equivalent if the corresponding cells within the

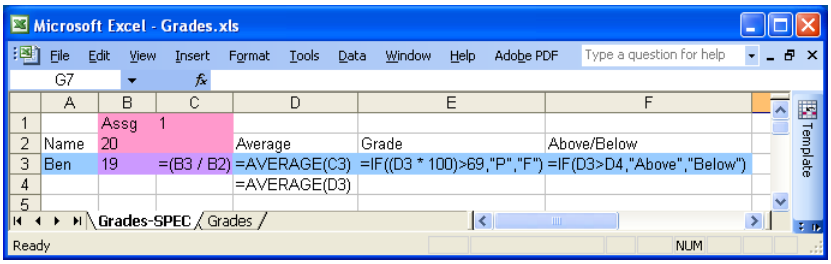


Figure 4.8: Template inferred from grade sheet.

columns are $\equiv_\eta$-equivalent.

$$S\lceil i\rceil \equiv_\eta S\lceil j\rceil \iff \forall k : C_S(i,k) \equiv_\eta C_S(j,k)$$

We can extend $\equiv_\eta$ one step further to regions within spreadsheets. A horizontal region can be formed from the repeated horizontal composition of adjacent columns of cells. We say that two regions are $\equiv_\eta$-equivalent if they both have the same number of columns and the corresponding columns are $\equiv_\eta$-equivalent. The importance of identifying equivalent regions lies in the fact that horizontally (or vertically) aligned equivalent regions can be *compressed* to a single instance in the template by repeatedly overlaying the equivalent regions onto the innermost copy.

The ViTSL template inferred from the grade sheet is shown in Figure 4.8. Columns B and C, in the template, form the hex group (shaded pink) for the assignment scores. Row 3 forms the vex group (shaded blue) for the student information. Cells B3 and C3 are in both the vex and hex groups (shaded purple). In the following, we describe the functions that carry out template inference using the grade sheet as the example.

Template inference is carried out in the following 5 main steps. First, the function

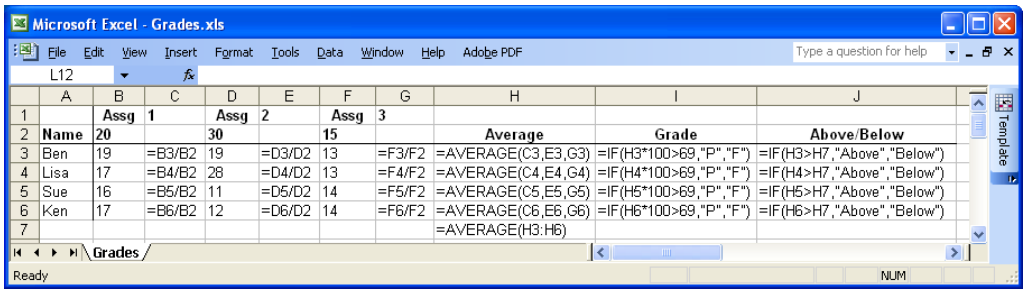
INFER$(S, \equiv_\eta)$ extends all the cells in the spreadsheet S by the result of the function η, on which the equivalence relationship $\equiv_\eta$ is based.

The function PATGEN$(\tilde{S})$ is then called on the extended spreadsheet $\tilde{S}$. The equivalence comparisons are all performed by the function "$=_\eta$", which simply compares the x components.

$$(a_1,(f_1,x_1)) =_\eta (a_2,(f_2,x_2)) \iff x_1 = x_2$$

Algorithm 1: Pattern Inference
Input: Spreadsheet S and the equivalence relation $\equiv_\eta$.
Output: Template for the spreadsheet S.
INFER$(S, \equiv_\eta)$
(1) $\tilde{S} \leftarrow \{(a,(f,x)) \mid (a,f) \in S, x = \eta(f)\}$
(2) $\tilde{S}' \leftarrow$PATGEN$(\tilde{S})$
(3) **return** $(\tilde{S}')$

Algorithm 2: Pattern Generation
Input: Spreadsheet S.
Output: Pattern p_x (which is S' with the number of repetitions marked).
PATGEN(S)
(1) $G \leftarrow S/=_\eta$
(2) $S' \leftarrow S$
(3) **while** $G \neq \emptyset$
(4) **foreach** $g \in G$
(5) $S_p \leftarrow$ MAXOVERLAY(g)
(6) **if** $S_p = ()$ **then continue**
(7) **else**
(8) $(S\lceil i \rceil, S\lceil j \rceil) \leftarrow S_p$
(9) $S'\lceil i \rceil \leftarrow$ OVERLAY(i,j)
(10) **if** $S = S'$ **then return** S
(11) **else**
(12) $G \leftarrow S'/=_\eta$
(13) $S \leftarrow S'$
(14) **return** S'



	A	B	C	D	E	F	G	H	I	J
1		Assg	1	Assg	2	Assg	3			
2	Name	20		30		15		Average	Grade	Above/Below
3	Ben	19	=B3/B2	19	=D3/D2	13	=F3/F2	=AVERAGE(C3,E3,G3)	=IF(H3*100>69,"P","F")	=IF(H3>H7,"Above","Below")
4	Lisa	17	=B4/B2	28	=D4/D2	13	=F4/F2	=AVERAGE(C4,E4,G4)	=IF(H4*100>69,"P","F")	=IF(H4>H7,"Above","Below")
5	Sue	16	=B5/B2	11	=D5/D2	14	=F5/F2	=AVERAGE(C5,E5,G5)	=IF(H5*100>69,"P","F")	=IF(H5>H7,"Above","Below")
6	Ken	17	=B6/B2	12	=D6/D2	14	=F6/F2	=AVERAGE(C6,E6,G6)	=IF(H6*100>69,"P","F")	=IF(H6>H7,"Above","Below")
7								=AVERAGE(H3:H6)		

Figure 4.9: Sample grade sheet.

To infer the template for a given spreadsheet, we first partition columns into $\equiv_\eta$ equivalence classes, that is, we compute $S/{=_\eta}$. To explain with an example how this works, we use the cp-similarity criterion of cells for $\equiv_\eta$. (The notion of cp-similarity for the cells has been explained in Section 2.3.) In the case of Parcel, $\eta(f)$ yields the *R1C1*-style representation of the formula f in a cell. We follow the approach described in [31] and decide two formulas are cp-similar by comparing their *R1C1*-style representations. The given definition of formulas makes the concept of cp-similar cells simple: Two formulas are cp-similar if and only if their *R1C1*-style representations are equal. For example, in the grades spreadsheet, the formulas in C3 (=B3/B2) and C4 (=B4/B2) in the *R1C1* style are both =RC[-1]/R2C[-1]. Therefore the formulas are cp-similar. For the grade spreadsheet shown in Figure 4.9 the formulas in columns C, E, and G are all =RC[-1]/R2C[-1]. Therefore these three columns are in the same equivalence class (under the cp-similarity criterion) and can potentially be overlaid.

For each group g of columns in an equivalence class, we determine, by the call to MAXOVERLAY(g), the columns that we can overlay that would in turn result in the biggest regions being overlaid. If an overlay is not possible for the group under con-

sideration, we pick the next group, and the overlays are carried out whenever possible. Note that every time one region is overlaid on another, we increment a counter at the column level to keep track of the number of times a column instance has been compressed. This information is implicitly stored in S'. Therefore, the call to PATGEN(S) returns S' along with the number of repetitions of the columns in S', that is p_x.

MAXOVERLAY(g) takes an equivalence group g of columns and returns the indices of the two columns in g that are furthest apart and can be overlaid. For an overlay to be successful, it must meet the following condition

$$\forall k \in \{0,...,d\} : S\lceil i-k \rceil \equiv_{\eta} S\lceil j-k \rceil$$

where $d = j - i - 1$. That is, $S\lceil j \rceil$ can be overlaid on $S\lceil i \rceil$ if and only if the columns between $S\lceil i \rceil$ and $S\lceil j \rceil$ can be overlaid on the corresponding columns before $S\lceil i \rceil$. If no overlay is possible for g, MAXOVERLAY(g) returns the null tuple.

Considering the three columns (C, E, and G) in the grade spreadsheet that are in the same equivalence class, overlaying E on C would require that we also overlay D on B. Similarly, overlaying G on C would also require that we overlay F on B. Since the cells in columns B, D, and F only have data values (no formulas) and the types agree, the overlays are possible.

In the grade sheet example, the system performs the following steps to overlay columns D and E onto columns B and C, respectively.

1. Only the formulas in column H have references to one of the columns being overlaid (column E in this case). The system updates the formulas in column H to

remove these references by a call to UPDATEFORMULA$(S,8,8,5,5)$. For example, the formula in H3 is modified to =AVERAGE(C3,G3).

2. Columns D and E are removed, thereby shifting all columns to their right two columns to the left. Note that SHIFT(S,m,n,d) also adjusts references to cells that shifted to reflect their new positions. For example, when the shifting moves cell I3 to its new position at G3, the cell formula is changed from =IF(H3*100>69,"P","F") to =IF(F3*100>69,"P","F") to reflect the fact that cell H3 has been shifted to F3.

3. Steps 1 and 2 are now repeated to overlay the new columns D and E (old positions were F and G) onto B and C.

Once the above steps have been completed, there are no more candidates for hex groups.

Algorithm 3: Maximal overlay in an equivalence group.
Input: Equivalence class g.
Output: The column numbers l and m that result in the maximal overlay.
MAXOVERLAY(g)
(1) $d_m \leftarrow 0$
(2) $l \leftarrow 0$
(3) $m \leftarrow 0$
(4) **foreach** $S\lceil i \rceil \in g$
(5) **foreach** $S\lceil j \rceil \in g \wedge j \neq i$
(6) **if** $j > i$ **then** $d \leftarrow j - i$
(7) **else** $d \leftarrow i - j$
(8) **if** $\forall k \in \{0,...,d\} : S\lceil i-k \rceil =_{\eta} S\lceil j-k \rceil$
(9) **if** $d > d_m$
(10) $d_m \leftarrow d$
(11) $l \leftarrow \min(i,j)$
(12) $m \leftarrow \max(i,j)$
(13) **if** $l = 0 \vee m = 0$ **then return** $()$
(14) **else return** (l,m)

OVERLAY(i, j) takes the two columns that have to be overlaid (as computed by MAXOVERLAY(g)) as input and returns the resulting sheet S'' after carrying out the overlay. As described above for the grade sheet example, before overlaying one column over another, we first need to update the formulas in the cells in the region $S\lceil j+1..j'\rceil$ by the call to UPDATEFORMULA$(S, j+1, j', i+1, j)$ to remove references to cells within region $S\lceil i+1..j\rceil$. This transformation of the formulas is subject to the constraint that references to cells within repeated groups from outside the repeated group have to be in aggregation formulas because we do not know beforehand how many times the region might be repeated. Since this update can only be done on aggregation formulas, overlaying fails (and will not be performed) if some formula in the region $S\lceil j+1..j'\rceil$ cannot be updated. In the definition of OVERLAY(i, j), $\overrightarrow{S}$ computes the index of the rightmost column in spreadsheet S, and in the definition of UPDATEFORMULA(S, m, n, q, r), $\downarrow S\lceil m..n\rceil$ computes the index of the lowermost row in the block $S\lceil m..n\rceil$.

Algorithm 4: Overlaying two columns from the same equivalence class.
Input: Spreadsheet S and column numbers i and j where $i < j$.
Output: Resulting spreadsheet S' after overlay.
OVERLAY(S, i, j)
(1) $j' \leftarrow \overrightarrow{S}$
(2) $d \leftarrow j - i$
(3) $S' \leftarrow$ UPDATEFORMULA$(S, j+1, j', i+1, j)$
(4) $S'' \leftarrow$ SHIFT$(S', j+1, j', d)$
(5) **return** S''

Finally, we need to shift the cells in region $S\lceil n+1..m\rceil$ to reflect the overlay, that is, each cell in the updated spreadsheet S' is mapped to a cell in the original spreadsheet S by SHIFT(S, m, n, d).

We repeat this process until there are no more $\equiv_\eta$-equivalent columns that can be

Algorithm 5: Updating formulas within a set of columns. The actual modifications to the individual formulas are done by the function update.
Input: Spreadsheet S and column numbers m, n, q and r.
Output: Resulting spreadsheet S' in which formulas in $S\lceil m..n\rceil$ have been updated to remove references to cells in $S\lceil q..r\rceil$.
UPDATEFORMULA(S,m,n,q,r)
(1) $k \leftarrow \downarrow S\lceil m..n\rceil$
(2) $S' \leftarrow S$
(3) **for** $i = m$ **to** n
(4) **for** $j = 1$ **to** k
(5) $C_{S'}(i,j) \leftarrow \mathsf{update}(C_S(i,j),q,r)$
(6) **return** S'

Algorithm 6: Adjusting columns after overlay.
Input: Spreadsheet S, the start (m) and end (n) positions of the columns to be adjusted, and the width (d) of the region being overlaid.
Output: Resulting spreadsheet S' after column adjustment.
SHIFT(S,m,n,d)
(1) **for** $i = 1$ **to** $m-d-1$
(2) $S'\lceil i\rceil = S\lceil i\rceil$
(3) **for** $i = m-d$ **to** $n-d$
(4) $S'\lceil i\rceil = S\lceil i+d\rceil$
(5) **return** S'

overlaid and we are left with the compressed sheet S'. We then carry out a similar process and compress the spreadsheet S' vertically by considering the $\equiv_\eta$-equivalent rows that can be overlaid. We are left with the pattern p_x which is the compressed form of the original spreadsheet S.

4.4 Evaluating the Quality of Inferred Templates

A given spreadsheet can potentially be generated from many different templates. Because of this high level of ambiguity in template inference, it is not possible to auto-

matically validate the correctness of a template inferred by Parcel. The creator of the spreadsheet would be the one in the best position to judge if the template generated by Parcel is an accurate summary of the original spreadsheet. This claim is based on the assumption that the creator of the spreadsheet has a good understanding of both, spreadsheets and ViTSL. We assume that this assessment would become more accurate with increasing experience with spreadsheet systems, the problem domain the spreadsheet was developed for, and the modeling language (ViTSL in this case). For example, an accountant with considerable experience with spreadsheets would be in better position to judge the correctness of a template for an accounting sheet than a person without any background in accounting. Even so, two accountants (with the same level of expertise within their domain and spreadsheets) might not necessarily agree on the same template as the correct one for a given spreadsheet.

Since there is no absolute measure of quality of templates, we decided to evaluate Parcel by comparing the templates generated by the system against those generated by novice and expert subjects. We were also interested in how experts and novices actually go about the task of manually inferring spreadsheet templates. This information would be useful for providing better tool support for the task within the Parcel user interface. Formally, we designed the study to be able to answer the following research questions.

RQ1: *How well does the system perform compared to expert and novice test subjects in extracting templates from spreadsheets?*

RQ2: *Are there any patterns of behavior exhibited by novice and expert subjects when they are trying to understand spreadsheets in order to develop their templates?*

4.4.1 Experiment Design

As novice subjects, we recruited 19 students from a course on Software Engineering at Oregon State University. We refer to this group of subjects as Group N. The course primarily dealt with the specification and design of software. UML was presented as the *de facto* standard modeling language for software, and ViTSL was presented as a language for modeling spreadsheets. Prior programming experience ranged from two to ten years (in two to four languages) and all the participants had between two and eight years of experience using spreadsheets. We chose students from this course since the target audience for ViTSL are people with beginning to intermediate level of programming and spreadsheet expertise.

We also enlisted help from four doctoral students working in the area of Programming Languages to serve as expert subjects. These subjects had five to ten years of programming experience (in two to five programming languages) and many years of experience with spreadsheets. They all also have experience with specification languages as part of their Ph.D. studies. We refer to this group of subjects as Group E.

For the purpose of the study, we decided to use spreadsheet from the EUSES corpus [52]. The corpus has 4498 spreadsheets collected from various sources. Since Gencel is not really useful for spreadsheets that do not contain formulas, we first isolated the 1977 spreadsheets in the corpus that had formulas in them. We then randomly selected 29 spreadsheets (20 workbooks) from this set for the study tasks.

An overview of the experiment is shown in Figure 4.10. The spreadsheets selected for the study were randomly assigned to the members of Group N so that each person

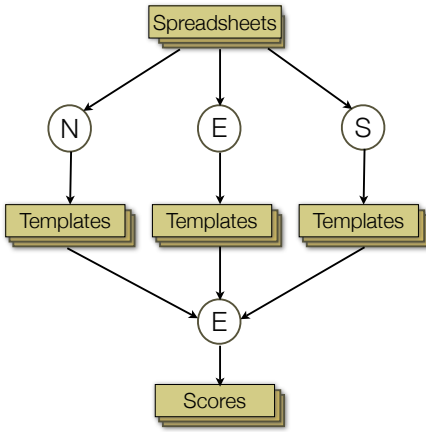


Figure 4.10: Overview of evaluation setup

was working on 5 to 6 spreadsheets. For each spreadsheet they were assigned, the participants were asked to sketch on paper (and provide a brief description of) the ViTSL template that could be used to generate it. We were hoping that the descriptions would be useful in cases in which the ViTSL templates developed by the participants were ambiguous or in cases where the participants were not comfortable with ViTSL. We made video recordings of the participants' interactions with the spreadsheets and later used the recordings for some of our analyses.

The subjects in Group E also went through the same exercise, inspecting the spreadsheets and sketching the ViTSL templates they could have been generated from. The 29 spreadsheets were randomly assigned to subjects in Group E such that each spreadsheet had two participants from the group working on them independently. Once again, we made video recordings of the subjects' interactions with the spreadsheets for post hoc analyses.

Score	Criteria
5 points	Spreadsheet can be generated from the template by insert/delete row/column commands and data updates exclusively
4 points	Overall structure of the template is correct, and only data or references in formulas in the template are incorrect
3 points	Some parts of the template structure like a vex or hex group were missing
2 points	Subject showed some understanding of templates but misunderstood the spreadsheet and got the template wrong
1 point	Template does not make any sense

Table 4.2: Scoring criteria for templates

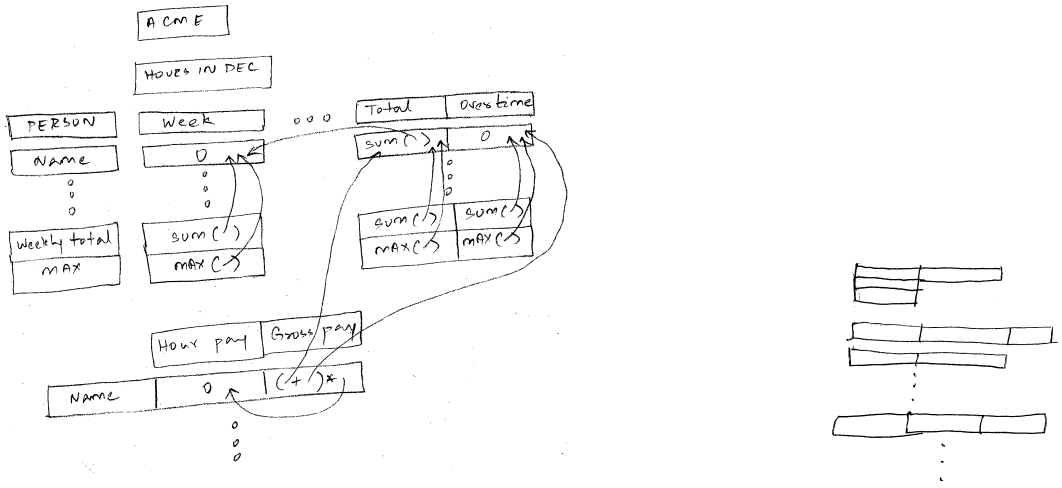
We did not impose any time limits on the subjects for the completion of the tasks. Two of the subjects from Group N stopped after an hour because of prior commitments and one stopped citing fatigue. All the other participants worked till the completion of their tasks. We also ran Parcel on the entire set of 29 spreadsheets (Group S). The ViTSL templates automatically inferred by the system were sketched on paper so that the final output would look similar to the work done by the participants from Group N and Group E.

All the templates were grouped together and then randomly assigned to the experts (ensuring that none of the experts graded their own work). The experts were instructed to grade the templates for correctness, based on the five-point scale shown in Table 4.2. They were not given any information about the subjects (or the groups) any template was developed by. As a matter of fact, the graders were not even aware that some of the templates had been generated by a system.

4.4.2 Threats to Validity

The threat to external validity is that the subjects in Group E are not really domain experts as far as the spreadsheets used in the study are concerned. However, these students have substantial programming and spreadsheet experience and are very familiar with ViTSL. Therefore, we think it quite safe to consider them experts as far as the experiment tasks are concerned. Moreover, it would be difficult (or even infeasible) to assemble a group of domain experts for a set of spreadsheets chosen randomly from a large heterogeneous corpus. One way to work around this problem would be to repeat the study within specific domains or organizations.

A threat to internal validity is the level of comfort of the subjects in groups N and E with templates and modeling languages (especially ViTSL). While the members of Group E have been exposed to ViTSL for over a year during research group meetings, presentations, and other discussions, the members of Group N were only exposed to ViTSL during the course. We have tried to minimize the impact of this factor by allowing the subjects to sketch the templates on paper (instead of making them work with the ViTSL template editor). They were instructed to try to capture the spreadsheet model in their sketch and descriptions rather than focusing on getting ViTSL syntax right. We also made it clear to the expert graders during the discussion of the grading criteria showed in Table 4.2 that the subjects were not to be docked points for not using correct ViTSL syntax.



(a) Detailed template

(b) Rough template

Figure 4.11: Inferred templates

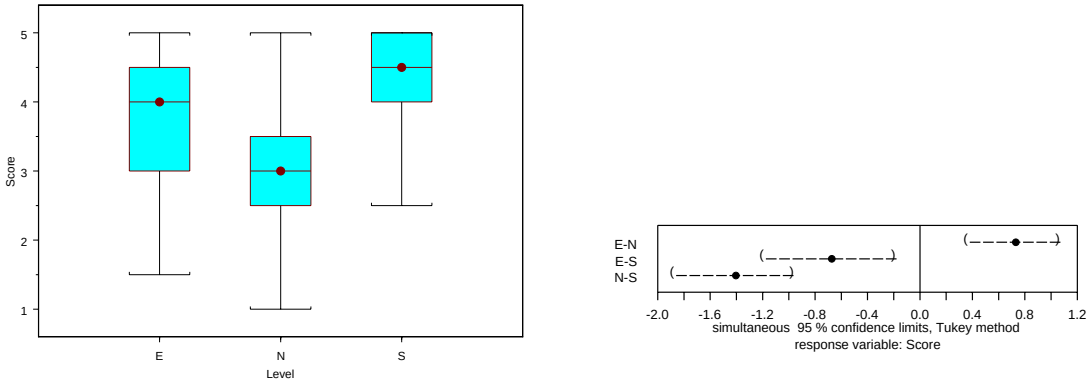
4.4.3 Results

In Figure 4.11 we show two examples of templates developed by the subjects. The template in Figure 4.11(a) shows the groups of cells, vertical and horizontal repeating groups, and relative references (as arrows since there is no global addressing scheme). The subject has even gone so far as to document the actual functions used in the cell formulas. The template shown in Figure 4.11(b), on the other hand, captures very little information about the corresponding spreadsheet. The template conveys some indication of cell blocks and vertically-repeating groups. Obviously, this template does not make much sense and requires a lot of work before it can be built in the ViTSL editor and used with Gencel.

Each template was rated by two experts based on the guidelines in Table 4.2. To compare the experts (A, B, C, and D), we determined the Kappa (κ) values for the rating

Graders	κ
A–B	0.76
A–D	0.71
B–C	0.70
C–D	0.74

Table 4.3: κ values for pairs of graders



(a) Task scores

(b) Three-way comparison of scores

Figure 4.12: Score comparisons

tasks on which different experts worked together to see how well their ratings agree. The κ values for the pairings of the graders are shown in Table 4.3, and all of them are greater than 0.6. Therefore, the agreement between the graders is good enough.

The boxplots comparing the average of the scores from the two graders for the different groups E, N, and S are shown in Figure 4.12(a). A pairwise comparison of the scores using the Tukey method is shown in Figure 4.12(b), and we see that none of the 95% confidence intervals include 0.

To answer **RQ1**, which dealt with the performance of the system when compared with the subjects in groups N and E, we carried out the following analyses of the data

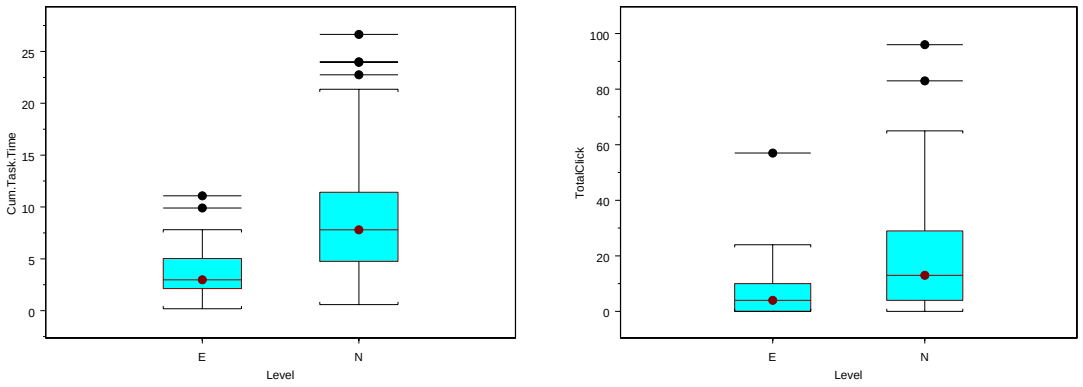
we collected.

System versus Group N The scores of the system-generated templates for the spreadsheets were significantly better than the scores of the templates developed by the subjects in Group N (ANOVA: $F(1,149)=51.69$, $p<0.001$). This result shows that the system is more reliable than the subjects with intermediate level of programming and spreadsheet experience.

System versus Group E We also compared the scores of the system-generated templates against the scores of the templates developed by the subjects in Group E. We expected the subjects in Group E (the experts) to perform better than the system since they have considerable programming and spreadsheet experience. Instead, we were surprised to find that the system performed significantly better than the subjects in Group E (ANOVA: $F(1,85)=11.75$, $p<0.001$). One possible explanation for this result could be that the spreadsheets in the study were too simple for the experts to outperform the system.

Group N versus Group E It is reasonable to assume that the expert subjects would outperform the novice ones on the assigned tasks. We compared the scores obtained by subjects in Group N against those obtained by the subjects in Group E just to confirm that this is the case. We see that that the subjects in Group E performed significantly better than those in Group N (ANOVA: $F(1,179)=22.17$, $p<0.001$).

Once we determined that the system outperformed both the experts and novices in the experiment tasks, we carried out post-hoc analysis of the video recordings of the



(a) Time taken (per spreadsheet) (b) Sheet inspection profile (per spreadsheet)

Figure 4.13: Subject behavior on tasks

subjects' interactions with the spreadsheets. The goal of this analysis was to identify any common patterns of behavior exhibited by the subjects while carrying out their tasks.

The analysis of the video recordings showed that the subjects in Group N spent significantly more time (boxplot shown in Figure 4.13(a)) on the tasks compared to subjects in Group E (ANOVA: $F(1,132)=32.82$, $p<0.001$).

There was no significant difference between the *inspection profiles* (number of cells inspected by the subjects while inferring the templates for a spreadsheet) of the subjects in Group N and Group E. The boxplot is shown in Figure 4.13(b). This result was surprising since we expected that the experts would need to inspect fewer cells to be able to infer the template for a given spreadsheet. The absence of significant difference in the inspection profiles might be because the experts were extremely cautious while carrying out their assigned tasks.

Microsoft Excel has an auditing tool that displays the data flow between cells in a

spreadsheet. Surprisingly, none of the subjects used this feature while inferring templates, even though all of them mentioned in the post-task interview that they were aware of it. Instead, the subjects relied more on cell labels and code inspection to locate formula cells while inferring templates.

To verify if the subjects found it more difficult to infer templates for bigger spreadsheets than for smaller ones, we ran regression tests comparing their scores on the tasks against the size of the spreadsheets. We found no significant correlation between scores and size of the spreadsheets for the two groups. This result is not surprising since the size of the spreadsheet is not really a good measure of its complexity. The number of formulas is also not a good measure since very simple templates can be used to generate large spreadsheets whose cp-similar regions have similar formulas. In such situations, humans might be able to infer the templates very accurately through visual inspection of the spreadsheet. A more reliable measure would be the number of distinct formulas (excluding cp-similar ones) within the spreadsheet.

We see from the statistical analysis that the templates inferred by Parcel score significantly higher than subjects in groups E and N. The mean time taken by the subjects in groups E and N to infer templates are 3.8 minutes and 3.9 minutes, respectively. If the time taken by Excel to load each spreadsheet is ignored, the system takes less than a second per spreadsheet to automatically infer the template. Therefore, the evaluation indicates that Parcel works well in practice and gives huge savings in terms of time and effort for users who wish to carry out template inference.

4.5 Error Detection Through Template Inference

There are two aspects of template inference that can be varied for different kinds of templates and region analysis.

1. The template specification language that is the target of template inference.
2. The equivalence condition used to determine regions that can be overlaid.

The specification language used by the current version of Parcel is ViTSL. As mentioned in Chapter 3, ClassSheets is another specification language developed for spreadsheets [42]. ClassSheets is more expressive (with close links to UML class diagrams) than ViTSL and could be the target of future reverse-engineering efforts. If the specifications have to be viewed by the user at any point, we also need to evaluate the usability of the target specification language.

The algorithms described in Section 4.3 do not enforce any restrictions on the equivalence condition used in template inference. Parcel uses cp-similarity, which is a strictly stronger criterion than type equivalence since cp-similarity implies type equivalence. However, in the context of spreadsheets, we can identify a stricter notion of cell types and type correctness which is influenced by the arrangement of cells within the grid structure of spreadsheets [6]. In our work described in [6] we used the refined notion of type equivalence for template inference, and found it to be a stronger criterion than cp-similarity as illustrated by the following example.

In Figure 4.14, we show the results of template inference using a combination of cp-similarity and type equivalence, on an erroneous version of the spreadsheet from Figure 4.9. The spreadsheet in Figure 4.9 has the formula IF(H3>H7,"Above","Below") in

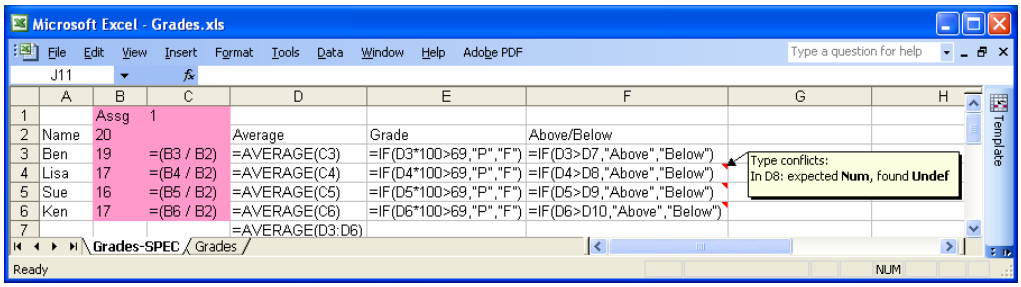


Figure 4.14: Template inference using spreadsheet types.

cell J3, and the formulas in the subsequent rows also compare the student's average with the class average in cell H7. In the erroneous spreadsheet the template in Figure 4.14 was inferred from, the reference to H7 was correct in the formula in H3. However, the subsequent rows have incorrect references to empty cells due to copy-paste errors. The hex group is correctly identified and shaded pink. Using Parcel with just cp-similarity as the equivalence criterion in this case would have missed the error in the original sheet during inference of the vex group since cells F3 through F6 all have the formula IF(RC[-2]>R[4]C[-2],"Above","Below"). As shown in Figure 4.14, the type-equivalence criterion flags the type errors in formulas in F4 through F6.

4.6 Conclusions and Future Work

In this chapter, we have described the ViTSL/Gencel framework that allows users to design spreadsheet templates and then use the templates to generate spreadsheets that are free from reference, range, and type errors. An integral part of the framework is the Parcel system that infers templates from spreadsheets. Flaws in the design of the

template might manifest themselves later as the user works with the spreadsheet in Gencel. The availability of Parcel makes it easier for the user to make modifications at the spreadsheet level and then infer a new template.

It is important to note that the guarantees provided by the ViTSL/Gencel framework are based on the assumption that the template being used by the user is correct. In future work, it would be beneficial to study ways in which the user can evaluate the correctness of any template they have created. A system that takes a template as input and automatically generates instances from it so the user can see how the spreadsheet would evolve under Gencel might be helpful in evaluating the correctness of the template.

The ViTSL/Gencel framework is the first attempt at capturing formal spreadsheet specifications and using them to generate spreadsheets that conform to the specifications. Parcel uses cp-similarity as the equivalence relationship to infer ViTSL templates from spreadsheets. As shown in [6], different classes of spreadsheet templates can be defined and inferred using different equivalence relationships on cells.

Tools that infer regions within spreadsheets using similarity conditions could help spreadsheet programmers identify errors. How this approach would work is by highlighting outliers within—what the programmer would assume to be—a homogenous region. For greater effectiveness, the user should also be able to combine (for example, union, intersection, difference, etc.) different visualizations depending on the kinds of errors they are trying to identify. Deviations from expected template, or presence of outliers within regions that should be homogenous, potentially indicate errors within the spreadsheet. The information could also be useful for someone carrying out an audit of the spreadsheet.

Chapter 5 – Automatic Consistency Checking for Spreadsheets[1]

If I had better equipment, I could do more damage.

Ashleigh Brilliant

Even though the benefits of type systems are well known, they are hard to use, even in general-purpose programming languages, primarily because the error messages generated by the type checkers are hard to understand. As a result of this problem, many programmers (especially beginners) find type systems more of a hinderance than help. The cause of the type error might not be obvious from the error messages, thereby making it difficult for the programmer to correct the error. This problem poses an even greater challenge for type systems targeted at end-user programming languages. The effectiveness of type checkers in preventing errors has been shown for general-purpose programming languages [96]. However, the effectiveness of type checkers for end-user languages has not been investigated so far. The main challenge in developing tools for end-user programming environments is to strike the right balance between formalism and usability.

Consider the spreadsheet shown in Figure 5.1. Cell B5 contains the number 5. The column and row headers Apple and July tell us that the number 5 is the number of apples from July. Similarly, the number 7 in cell C4 is the number of oranges from June. Clearly, the headers entered by the user in the spreadsheet give us additional

[1] The contents of this chapter are based on [1, 3, 8, 11].

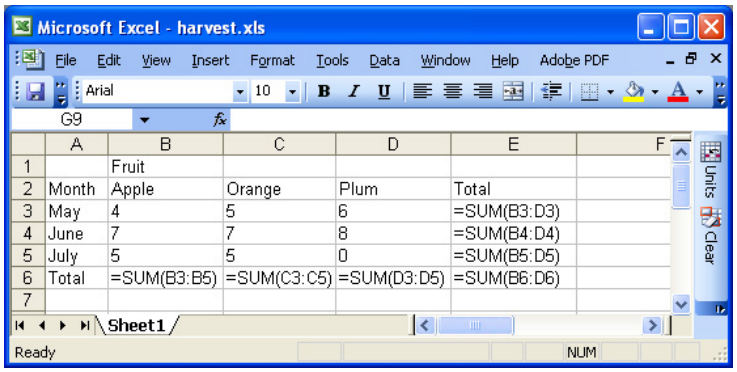


Figure 5.1: Harvest spreadsheet

information about the data stored in it. Studies have shown that in up to 85% of the cases, entering headers is the first thing users do when creating a spreadsheet [80]. Erwig and Burnett proposed using the headers entered by users in spreadsheets as implicit type declarations and using this information to carry out consistency checking of the spreadsheet formulas [45] to avoid the problem of cryptic error messages. Instead, in the this approach, the error messages could be in terms of the headers the users themselves had entered. Based on this idea, we have developed an automatic consistency checker called UCheck. This system is composed of two parts.

1. A formal reasoning system for detecting unit errors [45].
2. A header inference system that automatically determines the headers within the spreadsheet [1].

Using the headers entered by the user to carry out automatic consistency checking of spreadsheet formulas has the following two advantages.

1. The user does not have to provide any additional information for the system to

work since the headers are automatically inferred.

2. The inferred headers can be used to report inconsistencies within the spreadsheet using terminology the user is familiar with.

In the next section we describe the structure of units, and how they can be inferred and used for consistency checking once the headers are known. In Section 5.2, we describe the heuristics we have developed to infer the headers automatically. We evaluated UCheck by running it on a sample of spreadsheets. The results from this evaluation are presented in Section 5.3. In Section 5.4, we describe a think-aloud study we carried out to evaluate UCheck and present results from the study. We present conclusions and discuss directions for future research in Section 5.5.

5.1 Unit Inference

Units are not flat structures; rather, they exhibit a hierarchical structure. In this section, we describe the different kinds of units that can be inferred, the formal rules for unit combinations, and conclude with an example of how unit checking works in practice.

5.1.1 Unit Expressions

Unit expressions are built to reflect the spatial arrangement of cells and the functions used in the spreadsheets.

Base Units. In Figure 5.1, cell B5 contains the number 5 which has Apple and July as headers. The immediate header of a cell is its *base unit*. In the case of B5, Apple is

its column-level base unit and July is its row-level base unit. By convention, cells that do not have any headers have the base unit **1**. Therefore, the label Fruit in cell B1 has the base unit **1** since the cell does not have any headers.

Dependent Units. Since units are values, they can themselves have headers; hence, we can get chains of headers resulting in *dependent units*. Fruit is a header for Apple, Orange, and Plum. This hierarchical structure is reflected in our definition of units. In this example, the unit of the cell B3 is not just Apple, but Fruit[Apple]. In general, if a cell c has a value v as a unit which itself has header u, then c's unit is a *dependent unit* $u[v]$. Dependent units are not limited to two levels. For example, if we distinguished harvests from different states, a cell containing Apple could have the unit California[Fruit] (assuming California is one of the states the numbers are for), and the inferred dependent unit would be California[Fruit][Apple], which is the same as California[Fruit[Apple]].

And Units. Cells might have more than one unit. For example, the number 7 in cell C4 gives a number of oranges, but at the same time describes a number that is associated with the month June. Cases like this are modeled with *and* units. In our example, C4 has the unit Fruit[Orange]&Month[June].

Or Units. The dual to *and* units are *or* units. *Or* units are inferred for cells that contain operations combining cells of different, but related units. For example, cell E3's formula is SUM(B3:D3), which is equivalent to B3 + C3 + D3. Although the units of B3, C3, and D3 are not identical, they differ only in one part of their *and* unit, Fruit[Apple], Fruit[Orange], and Fruit[Plum]. Moreover, these units differ only in the innermost part of their dependent units. In other words, they share a common prefix that includes the complete path of the dependency graph except the first node. This fact makes the + operation

applicable. The unit of E3 is then given as an *or* unit of the units of B3, C3, and D3, that is, Fruit[Apple]&Month[May]|Fruit[Orange]&Month[May]|Fruit[Plum]&Month[May].

Not all unit expressions are meaningful. For example, a number cannot represent apples *and* oranges at the same time, although a number can represent apples *or* oranges *or* plums, that is, fruits. The rules of the unit system define the combination (and simplification) of unit expressions for formulas. Those formulas for which the unit system cannot derive a meaningful unit expression are considered to be incorrect in the sense that they contain a unit error. The following rules define all meaningful unit expressions.

1. Every value that does not have a header is a well-formed unit. For example, in Figure 5.1, Fruit is a well-formed unit.
2. If a cell has value v and header u, then it $u[v]$ is a well-formed unit. For example, in Figure 5.1, Fruit[Apple] is a well-formed unit.
3. Where there is no common ancestor, it is legal to *and* units. For example, in Figure 5.1, the unit of B3, Fruit[Apple]&Month[May] is a well-formed unit because Apple and May have no common ancestor. In contrast, the unit Fruit[Apple]&Fruit[Orange] is not well-formed.
4. Where there is a common header ancestor, it is legal to *or* units. For example, in Figure 5.1, Fruit[Apple]|Fruit[Orange]|Fruit[Plum], which denotes the same unit as Fruit[Apple|Orange|Plum], is well-formed. More precisely, we require that all the values except the most nested ones agree. This is the reason why the unit Fruit[Apple[Green]]|Fruit[Orange] is not well-formed.

5.1.2 A Formal Unit System

In this section we briefly describe the formal background of the unit system. The described unit system is based on the formal model first introduced in [45]. The formal rules for unit inference are shown in Figure 5.2. We need to consider three kinds of judgments: First, we have judgments of the form $(a, f) :: u$ that associate units to cells and that can exploit header information. Second, for the unit inference for operations we also need judgments $f : u$ that give units for expressions regardless of their position (context). Third, for expressing the units of a spreadsheet we need a further kind of judgment $(a_1, f_1); \ldots; (a_m, f_m) ::: (a_1, u_1); \ldots; (a_m, u_m)$, which expresses the fact that the unit information for a spreadsheet is given by a mapping of addresses to units. Note that the spreadsheet (S) and the header definition (h) are considered global constants in the rules.

The rules $\text{EQ}_{:}$ and $\text{EQ}_{::}$ are based on the equations shown in Figure 5.5, which define equality of units. Moreover, the premises $\vdash u$ ensure that only well-formed units, as defined in Figure 5.4, will be derived. For example, $\text{VAL}_{::}$ expresses that all cells that do not have a header have the unit $\mathbf{1}$. In the spreadsheet shown in Figure 5.1, cells Fruit and Month have unit $\mathbf{1}$. According to $\text{DEP}_{::}$, if cell a has a header cell b that contains a value v and has a well-formed unit u, then a's unit is $u[v]$. More generally, if a cell has multiple headers with values v_i and units u_i, then its unit is given by the *and* unit of all the $u_i[v_i]$. An example is the cell B3 containing 4 whose unit is Fruit[Apple]&Month[May]. $\text{REF}_{::}$ is applicable if cell a's formula is a reference to cell b. In this case, b's unit, say u_b is propagated to a. If a has itself a header, say u_a, then u_a must conform with u_b,

$$\text{VAL}_{:}\quad \frac{}{v : \mathbf{1}} \qquad \text{REF}_{:}\quad \frac{(a, S(a)) :: u \qquad \vdash u}{\uparrow a : u} \qquad \text{EQ}_{:}\quad \frac{f : u \qquad u =_u u'}{f : u'}$$

$$\text{APP}_{:}\quad \frac{f_i : u_i \qquad \vdash u_i}{\psi(f_1, \ldots, f_n) : \mu_\omega(\mathbf{1}, u_1, \ldots, u_n)}$$

$$\text{VAL}_{::}\quad \frac{a \notin dom(h)}{(a, v) :: \mathbf{1}} \qquad \text{REF}_{::}\quad \frac{(a, S(a)) :: u \quad (a', S(a')) :: u' \quad \vdash u \quad \vdash u'}{(a, \uparrow a') :: u \& u'}$$

$$\text{APP}_{::}\quad \frac{f_i : u_i \qquad \vdash u_i \qquad (a, \sqcup) :: u \qquad \vdash u}{(a, \psi(f_1, \ldots, f_n)) :: \mu_\omega(u, u_1, \ldots, u_n)} \qquad \text{EQ}_{::}\quad \frac{(a, f) :: u \quad u =_u u'}{(a, f) :: u'}$$

$$\text{DEP}_{::}\quad \frac{h(a) = \{a_1, \ldots, a_k\} \qquad S(a_i) = v_i \qquad (a_i, v_i) :: u_i \qquad \vdash u_i}{(a, v) :: u_1[v_1] \& \ldots \& u_k[v_k]}$$

$$\text{SHEET}_{:::}\quad \frac{i \in J_m \qquad (a_i, f_i) :: u_i \qquad \vdash u_i \qquad j \in \bar{J}_m \qquad u_j = \varepsilon}{(a_1, f_1); \ldots; (a_m, f_m) ::: (a_1, u_1); \ldots; (a_m, u_m)}$$

$$\text{PROXY}\quad \frac{S(b) = \psi(f_1, \ldots, f_n) \qquad h(b) = \{a\} \qquad (a, S(a)) :: u}{\wp(a, u)}$$

Figure 5.2: Unit inference rules.

which is achieved by defining a's unit to be $u_a \& u_b$. APP$_{::}$ is applicable in cases cells have formulas with operators. Each operator (μ_ω) has its own definition of how units of its parameters combine. The definitions of unit combinations are shown in Figure 5.1.2 for sum (μ_+), count (μ_{count}), product (μ_*), and division ($\mu_/$). (For the sake of convenience, we abbreviate $u_1, \ldots, u_n$ by $\bar{u}$ in Figure 5.3.) For example, the rule for μ_+ and μ_{count} state that those operations result in the units of the operands being combined together by &. PROXY is applicable to cells that have aggregation formulas. The $\wp(a, u)$ judgment expresses that the cell with address a is a proxy header for u. For example, in

the spreadsheet shown in Figure 5.1, cell E3 has the header Total (from cell E2) which is just a proxy for Fruit. This inference can be carried out since the aggregation formula in E3 references B3, C3, and D3, thereby including all the headers (Apple, Orange, and Plum) Fruit is the parent of.

$$
\begin{array}{ll}
\mu_{+}(u,\bar{u}) & = (u_1|\ldots|u_n)\&u \\
\mu_{\mathsf{count}}(u,\bar{u}) & = (u_1|\ldots|u_n)\&u \\
\mu_{*}(u,\bar{u}) & = \downarrow(\bar{u})\&u \\
\mu_{/}(u,\bar{u}) & = u_1\&u \\
\ldots &
\end{array}
\qquad
\downarrow(\bar{u}) = \begin{cases} u_i & \text{if } u_i \neq \mathbf{1} \wedge \forall j \neq i : u_j = \mathbf{1} \\ \mathbf{1} & \text{otherwise} \end{cases}
$$

Figure 5.3: Unit transformations.

Well-formed units are defined by the judgments in Figure 5.4. As in the case for the rules for unit inference, we assume that the spreadsheet (S) and the header information (h) are globally available. The shown rules formalize the conditions described earlier in Section 5.1.1.

$$
\text{ONE}_{\vdash}\ \frac{}{\vdash \mathbf{1}} \qquad \text{VAL}_{\vdash}\ \frac{a \notin dom(h) \qquad S(a) = v}{\vdash \mathbf{1}[v]}
$$

$$
\text{DEP}_{\vdash}\ \frac{a' \in h(a) \qquad S(a) = v \qquad \vdash u \qquad \vdash u[S(a')]}{\vdash u[S(a')[v]]}
$$

$$
\text{AND}_{\vdash}\ \frac{\vdash \mathbf{1}[u_1[\ldots u_{n-1}[u_n]\ldots]] \qquad \vdash \mathbf{1}[u'_1[\ldots u'_{m-1}[u'_m]\ldots]] \qquad u_i \neq_u u'_i}{\vdash u_1[\ldots u_{n-1}[u_n]\ldots]\&u'_1[\ldots u'_{m-1}[u'_m]\ldots]}
$$

$$
\text{OR}_{\vdash}\ \frac{\vdash \mathbf{1}[u_1[\ldots u_n[v_1]\ldots]] \ \ldots \ \vdash \mathbf{1}[u_1[\ldots u_n[v_k]\ldots]] \quad n > 1 \quad k > 1 \quad v_i \text{ distinct}}{\vdash \mathbf{1}[u_1[\ldots u_n[v_1|\ldots|v_k]\ldots]]}
$$

Figure 5.4: Well-formed units.

Application of operations can result in arbitrarily complex unit expressions that do

not always meet the conditions for well-formed units. Figure 5.5 shows the properties of well-formed units that allow us to simplify complex unit expressions. Whenever simplification to a well-formed unit is possible, it can be concluded that the operation is applied in a unit-correct way. Otherwise, a unit error is reported. Note that & distributes over |, but not vice versa, and that although **1** is the identity unit for &, it leads to a non-valid unit when combined with |. The condition (*) for generalization is that the *or* unit expression consists exactly of all non-proxy units $u_1, \ldots, u_n$ that have u as a header. Moreover, this condition must hold for all definitions (copies) of u (that is, for all cells containing u):

$$\forall a \in S^{-1}(u) : h^{-1}(\{a\}) = \{a_1, \ldots, a_k, a_{k+1}, \ldots, a_{k+m}\} \ \wedge \ S(a_i) = u_i \ (\text{for} \, 1 \leq i \leq k) \wedge$$

$$\wp(a_i, u) \ (\text{for} \, k+1 \leq i \leq k+m)$$

For example, cell E3 in Figure 5.1 has the unit Month[May]&Fruit[Apple|Orange|Plum]. This unit can be generalized to Month[May]&Fruit since the *or* unit expression has all units Apple, Orange, and Plum, that have Fruit as their header.

5.1.3 An Example for Unit Checking

We demonstrate the unit inference rules and the unit simplifications with the example spreadsheet from Figure 5.1 which we call *Harvest*. Since B1 has no header definition, Fruit has unit **1** by rule VAL$_{::}$, that is $(\mathsf{B1}, \mathsf{Fruit}) :: \mathbf{1}$. Similarly, the unit for Month in cell A2 is inferred as **1**. By rule DEP$_{::}$ we know that Apple has the unit **1**[Fruit], or

$$
\begin{array}{rll}
u_1 \& u_2 =_u u_2 \& u_1 & & \text{(commutativity)} \\
u_1 | u_2 =_u u_2 | u_1 & & \\
(u_1 \& u_2) \& u_3 =_u u_1 \& (u_2 \& u_3) & & \text{(associativity)} \\
(u_1 | u_2) | u_3 =_u u_1 | (u_2 | u_3) & & \\
u \& (u_1 | u_2) =_u (u \& u_1) | (u \& u_2) & & \text{(distributivity)} \\
u \& u =_u u & & \text{(idempotency)} \\
u | u =_u u & & \\
\mathbf{1} \& u =_u u & & \text{(unit)} \\
u[u_1] | u[u_2] =_u u[u_1 | u_2] & & \text{(factorization)} \\
u[u_1 | \ldots | u_k] =_u u & \Leftarrow (*) & \text{(generalization)} \\
(u_1[u_2])[u_3] =_u u_1[u_2[u_3]] & & \text{(linearization)}
\end{array}
$$

Figure 5.5: Unit equality.

$(\mathsf{B2}, \mathsf{Apple}) :: \mathbf{1}[\mathsf{Fruit}]$, because

- the header of B2 is B1,
- the value of the cell B1 is Fruit (that is, $\mathit{Harvest}(\mathsf{B1}) = \mathsf{Fruit}$), and
- cell Fruit has unit **1**.

Since all units except **1** are of the form $\mathbf{1}[\ldots]$ we omit the leading **1** for brevity in the following. Hence we also write $(\mathsf{B2}, \mathsf{Apple}) :: \mathsf{Fruit}$. Similarly, we can reason that $(\mathsf{A3}, \mathsf{May}) :: \mathsf{Month}$. Finally, using the last two results and the fact that the header of B3 is defined to be the two cells A3 and B2 we can conclude by rule $\text{DEP}_{::}$ that $(\mathsf{B3}, 4) :: \mathsf{Month}[\mathsf{May}] \& \mathsf{Fruit}[\mathsf{Apple}]$.

Next we infer the unit for the cell E3. First, we have to infer the units

$$(\mathsf{C3}, 5) :: \mathsf{Month}[\mathsf{May}] \& \mathsf{Fruit}[\mathsf{Orange}]$$

$$(\mathsf{D3}, 6) :: \mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Plum}]$$

which can be obtained in the same way as the unit for the cell B3. To infer the unit of E3 we apply rule $\text{APP}_{::}$. We already have the units for the three arguments. We can apply μ_+ and obtain as a result an *or* unit for E3, namely:

$$\mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Apple}]|\mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Orange}]|\mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Plum}]$$

We can now perform rule simplification several times, first exploiting distributivity, which yields the unit:

$$\mathsf{Month}[\mathsf{May}]\&(\mathsf{Fruit}[\mathsf{Apple}]|\mathsf{Fruit}[\mathsf{Orange}]|\mathsf{Fruit}[\mathsf{Plum}])$$

Then we apply factorization, which yields the unit:

$$\mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Apple}|\mathsf{Orange}|\mathsf{Plum}]$$

The aggregation formula in E3 has references to B3, C3, and D3, which have Apple, Orange, and Plum as their headers respectively. Total in E2 has Fruit as its unit. We can apply the PROXY rule to conclude that Total is simply a proxy header for Fruit and can be ignored in the condition for generalization. Therefore, using generalization, we obtain the following unit for E3

$$\mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}$$

Now consider a cell with an expression $+(\mathsf{B3}, \mathsf{C4})$. The unit for this cell would be inferred as $\mathsf{Month[May]\&Fruit[Apple]||Month[June]\&Fruit[Orange]}$. Since we get two different months and two different fruits, the application of the distributivity rule is not possible, which prevents the unit expression from being reduced to a well-formed unit. Since we cannot infer a unit in normal form, the system reports a unit error for that cell.

5.2 Header Inference

Automatic header inference is crucial to minimize the effort required of the user to carry out consistency checking. It is the link to a completely automatic unit checker and differentiates UCheck from the other systems described in Sections 3.3.1, 3.3.2, and 3.3.3 that require the user to manually annotate the spreadsheet cells with additional information to facilitate consistency checking.

5.2.1 Header-inference Framework

We exploit the spatial layout of the spreadsheet cells to carry out header inference. Since spreadsheets differ greatly in their layout, it is unlikely that a single algorithm works equally well in all cases. Therefore, we developed a header-inference framework that allows the integration of different heuristics for spatial analyses.

Header inference is based on the view that a spreadsheet is essentially composed of one or more *tables*. We use the information about the spatial arrangement of cells to classify the cells in a spreadsheet into the following groups.

1. Header: The user uses these to label the data.
2. Footer: These are typically placed at the end of rows or columns and contain some sort of aggregation formula.
3. Core: These are the data cells.
4. Filler: These can be blank cells or cells with some special formatting used to separate tables within the sheet.

We have defined several algorithms that classify spreadsheet cells into the categories mentioned above. Since the algorithms are not equally accurate at identifying the roles of the cells, we assign levels of confidence to the classifications depending on the algorithm used. The confidence levels can range from 1 (minimum confidence) to 10 (maximum confidence). The header inference framework has been designed to allow the easy selection of any combination of algorithms and the weights used by them. Whenever a cell is classified in multiple categories, we sum the confidence levels for each of the categories and pick the classification with the highest sum. This flexibility has allowed us to study the performance and effectiveness of the individual algorithms and in tuning the confidence parameters associated with the algorithms.

There are different ways of classifying the cells within the spreadsheet based on their roles. In the absence of errors in the spreadsheet, a preliminary classification of the cells based on their roles is conceivable. We can assign cell roles by analyzing the formulas as follows.

- Cells that have formulas and are not referred to by other cells can be classified as footers.

- Cells that do not have formulas and are used as inputs to other cells can be classified as core cells.
- Cells that neither have formulas nor are used as inputs to formulas in other cells can be considered headers.
- Cells that have formulas that reference core cells or other formula cells can be classified as footers.

This rather simplistic classification of the spreadsheet cells fails when we have errors in the spreadsheet formulas. That is why we have adopted a heuristics-based approach. The following strategies are employed to classify spreadsheet cells.

Fence Identification. A *fence* is a row or a column of cells that form a boundary (upper, lower, left, or right) of a table. If the fence consists of blank cells, we treat it as a *hard* fence, otherwise, we treat it as a *soft* fence, which is typically the case when the fence consists of repeated headers. Hard fences are classified with a high level of confidence, whereas soft fences are classified with a lower level of confidence.

Content-Based Cell Classification. This algorithm classifies cells as headers, footers, and core simply based on their content. For example, cells with aggregation formulas are classified as footer cells, cells with numerical values are classified as core cells, and cells with string values are classified as header cells. The classification performed by this algorithm is assigned a low level of confidence.

Region-Based Cell Classification. Whenever we have knowledge about the extent of a table, which is the case, for example, after we have identified fences, we can classify some roles with a higher level of confidence. For example, if the top row or leftmost column of a table is composed of strings, we classify them as headers with a high level

of confidence. Similarly, if the last row or rightmost column of a table has aggregation formulas, we classify these as footers.

Footer-to-Core Expansion. In a first step we identify the cells that have aggregation formulas. Such cells are classified as footers with a low level of confidence. We then look at the cells that are referenced by the aggregation formulas and classify them as core cells with a high level of confidence. These cells are called *seed cells*. After that we look at the immediate neighbors of the seed cells. If they are of the same type as the seed cells, they are classified as core cells, too. In this way, we use the identified seed cells to grow the core regions. Once we have identified the core and footer cells, we can mark the rest of the cells as header or filler depending on whether or not they have content. This algorithm allows us to identify core cells, headers, and footers.

5.2.2 Header Assignment

After the classification has been carried out, we can assign headers to the core and footer cells. Header assignment is carried out in two steps.

1. As the first step, we assign the *first-level* headers for the core cells.
2. We then assign *higher-level* headers which are basically headers of the lower-level headers.

For every core cell, we assign as first-level headers the nearest row (to the left) and column (above) header cells. For example, this strategy results in cell B4 being assigned Apple and June as headers, see Figure 5.6. The other first-level headers will be Orange,

Plum, Total (in cell E2), May, July, and Total (in cell A6). Note that first-level headers cannot act across fences. In contrast, higher-level headers can act across fences.

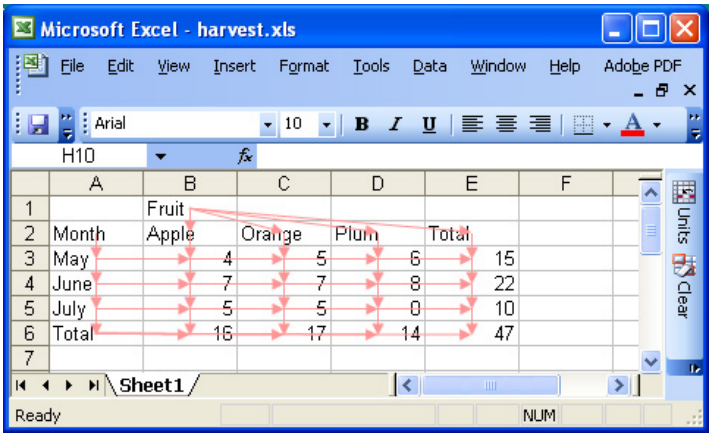


Figure 5.6: Inferred headers.

Once the first-level headers have been assigned, we can partition the original set of headers into two sets depending on whether or not they have already been assigned to a core cell. Let A be the headers that have been assigned to core cells and U be the set of headers that have not been assigned to any cell yet. In the example spreadsheet shown in Figure 5.6, Fruit and Month cannot be assigned as first-level headers and belong to set U. Some of the elements of set U might be candidates for higher-level headers whereas others might be just comments.

We impose the following restrictions while inferring higher-level headers.

1. An n-level header cannot be assigned as a header for another header at level n.

2. A header at level n can only have one header at level $n+1$. If this rule were violated, the resulting dependent unit would not be well-formed.

3. If two cells are headers for a core cell, they cannot have a common header assigned to them. For example, in Figure 5.6, the core cell B4 has been assigned Apple (cell B2) and June (cell A4) as headers. If B2 and A4 were assigned a common header, B4 would result in a unit error (violation of Rule 3 in Section 5.1.1).

Constraints 2 and 3 essentially limit headers to trees and prevent DAGs.

In addition to the above, we impose the following spatial constraints to exclude user comments from being inferred as headers.

1. We do not assign a higher-level header with only one child since such an assignment would not be of any use from a unit-inference point of view.
2. Because of the previous condition, if there are k headers at level n, level $n+1$ can have at most $k/2$ headers. We also require that the headers at level $n+1$ be separated by at least the average distance between the headers at level n. (We will discuss this distance measure in more detail below.)

Elements of set U that do not satisfy these constraints are excluded from the set. The headers in set A are either row headers or column headers. Any element $u \in U$ can potentially be a higher-level column header for a subset of column headers $\{a_i | a_i \in A\}$ if the row number of u is less than the row numbers of all a_i's. In other words, a higher-level column header has to be located at the same row level or above the cells it is the header of. Similarly, for higher-level row headers, we require that they are at the same column level or to the left of the cells they are a header of.

In addition to the above conditions, we have a *cost* associated with every assignment of some a_i to u. For column headers, the cost is the column distance between u and a_i

and in the case of row headers, the cost is the row distance between u and a_i. Unassigned elements in U receive an infinite cost to encourage the assignment of all valid higher-level headers. Once the system has generated all the possible combinations, it tries to minimize the overall cost. We first demonstrate how this works using the simple example in Figure 5.6 and then look at a more complicated case and discuss an extension to this algorithm.

In the example in Figure 5.6, Apple, Orange, Plum, and Total (in E2) are assigned as column headers and May, June, July, and Total (in A6) are assigned as row headers by the nearest-header algorithm discussed above. This leaves Fruit and Month as the only elements of set U. Fruit cannot be assigned as a row header for cells A3, A4, A5, and A6 because of the spatial constraints discussed above. The cost for assigning Fruit as header for Apple, Orange, Plum, and Total is $0+1+2+3=6$. In contrast, the cost for assigning Month as the column header for Apple, Orange, Plum, and Total (in E2) is 10. Only one of these assignments (either Fruit or Month) can be selected. If the assignment for Month is selected, Fruit will remain unassigned, which would result in an overall infinite cost for the assignment. On the other hand, the cost in assigning Month as row header for May, June, July and Total (in A6) is also 10. This assignment results in an overall cost of 16 since it does not conflict with the assignment of Fruit as column header. Therefore the systems assigns the headers as shown in Figure 5.6.

We consider a more complicated spreadsheet shown in Figure 5.7. The spreadsheet contains data for the number of cars exported to USA, Europe, and Asia, broken down by makers and years.

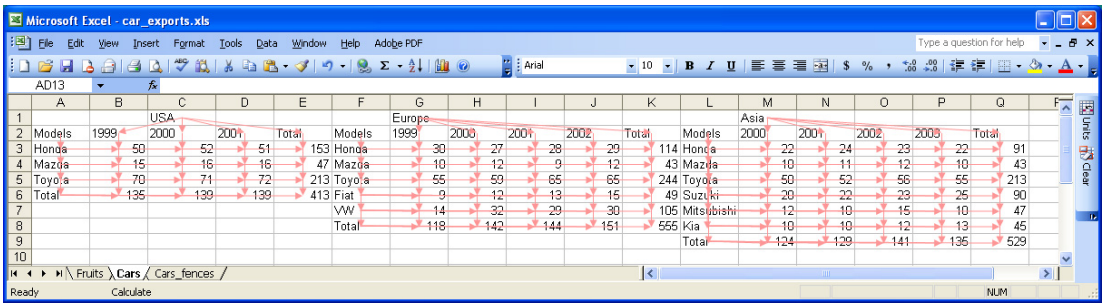


Figure 5.7: Car exports.

After the cell classification and the identification of first-level headers, Models is assigned as the second-level row header for the first-level headers under it in columns A, F, and L. To assign the column headers, we exploit the fact that a label can be expected to be in the proximity of (some of) the cells it is describing, because otherwise, it would not serve its purpose. Moreover, people have their own preferences in how they position higher-level headers. For example, in the case of column headers, some people might prefer to position the higher-level header above the first column of subheaders (as we have done in Figure 5.6 with Fruit), whereas others might prefer to position the higher-level header centered above its subheaders. In the current example, Europe and Asia have been positioned as per the first convention whereas USA has been more or less centered above its subheaders. The system takes advantage of the spatial information and fences to come up with the correct header inference shown in the figure in the following way.

1. First, we compute the column distance between USA and Europe. Let this distance be d_1 (4 in this case). Starting at cell B2 we traverse distance d_1 to the right

and reach cell F2. Models in A2 is ignored since it has already been assigned as a second-level header for the other cells in column A. The same is the case with columns F and L. Based on our discussion above, we consider $d_1 \pm 1$ as a good metric for the level 1 headers that should be made subheaders of USA. Similarly, we compute the column distance between Europe and Asia, say d_2 (6 in this case). We start at G2 and traverse distance d_2 to the right and reach cell M2. In this case, $d_2 \pm 1$ is a good metric for the level 1 headers that fall under Europe.

2. We have additional clues from the soft fences that are inferred. Since column E is composed of footer cells and column F is composed of header cells, we can infer these as soft fences. Similarly, we can infer soft fences for columns K and L. This information also helps us to correctly assign the higher-level column headers.

The resulting headers are as shown in Figure 5.7.

5.2.3 Optimizing Header Inference

In this subsection we describe how header inference works using a case study that demonstrates how the header inference framework helped us to integrate the different algorithms.

While testing the header inference system with the footer-to-core expansion algorithm running at confidence rating 4 and the content-based classification algorithm running at confidence rating 2 we could observe that the system inferred incorrect/insufficient headers for the example spreadsheet shown in Figure 5.8 that contains a range error. Since cell B2 participates in the aggregation formula in B5, the footer-to-

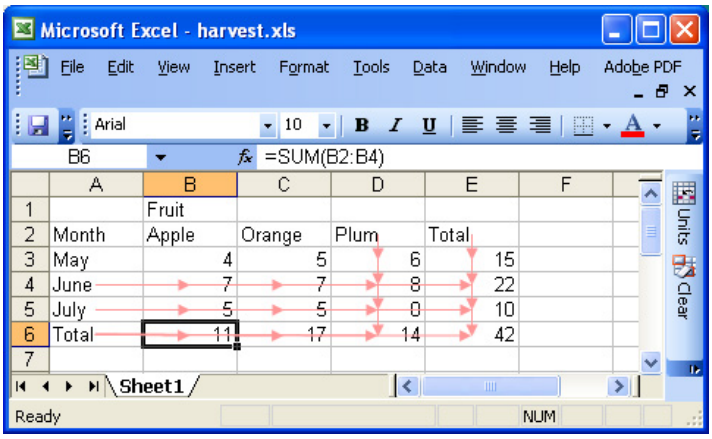


Figure 5.8: Unsatisfactory header inference.

core expansion algorithm marks it as a core cell. Furthermore, it checks the neighbors of B2 that have the same type and marks the cells A2, A3, B1, and C2 as core cells (with a confidence level of 4). Since these cells have string values, the content-based classification algorithm marks them as headers with a confidence level of 2. Running only these two algorithms results in the incorrect header inference shown in Figure 5.8. Increasing the confidence rating of the content-based classification algorithm to a value higher than that of the footer-to-core expansion algorithm would resolve the problem in this particular case, but it would be incorrect in general since the system would then always classify string values as headers.

When the region-based classification algorithm is also enabled, it classifies the first row as a soft fence (one non-blank cell) and the seventh row and column F as hard fences. This results in all the cells in the second row being classified as headers with a confidence rating of 3, the cells in first column being classified as headers with a confidence rating of 5, the cells with formulas in row 6 and column E being classified

as footers with a confidence rating of 5. When these classifications are combined with those discussed above, the system performs the correct header inference as shown in Figure 5.6.

We have arrived at the set of classification algorithms described in this section and their corresponding confidence ratings based on their performance over a wide variety of sample spreadsheets. The framework gives the developer the added flexibility of altering the set of classification algorithms used or their confidence ratings if the need arises.

UCheck works well in cases when headers are well organized into hierarchies and are placed in spatial proximity to the values they annotate. In fact, such a "labeling discipline" is partly assumed and exploited by the header inference. In practice we have observed that spreadsheets are annotated in such a way most of the time. This fact is not too surprising because users introduce labels to make sense of the numbers in the spreadsheet. To have this effect, labels must be placed in some structured and reasonable way. Consequently, strong deviation from this practice will cause UCheck to produce wrong results.

5.3 Preliminary Evaluation

We tested UCheck on two sets of spreadsheets. The first set (set A) consisted of 10 spreadsheet examples from a book by Filby [49] on spreadsheets in science and engineering. The second set (set B) consisted of 18 spreadsheets developed by first-year undergraduate students as part of an introductory course on programming. From among

all the sheets developed by the students as part of the coursework, we selected spreadsheets from each assignment, and also included those that had unusual spatial layout.

The header inference system incorrectly identified higher-level headers in some cases, especially with the sheets in set B. For example, in a couple of cases, the system identified the student's name and identification number as master column headers for the spreadsheet cells. Since the output from the header inference algorithms is fed to the unit system, incorrect header inference might result in the system reporting unit errors incorrectly. However, this did not happen with our system. Even in the few cases in which the system produced slightly incorrect headers, the unit inference did not report any illegal unit errors, because the header inaccuracies occurred for unimportant labels.

Regarding the accuracy of header inference, in set A our system incorrectly reported 4 headers in 1 sheet. In set B, the system reported 3 wrong headers in one sheet and 2 wrong headers in another sheet. As we have mentioned, in no case did an incorrectly inferred header lead to an illegal unit error.

Regarding unit inference, our system detected an omission error in one of the sheets of set A (in the worksheet "P-Cleavage" in workbook "ERTHSCI.XLS"). The same error was detected by the system developed by [15]. This shows that our automatic header inference system works as well as their system, which requires users to manually annotate the cells with unit information. Since this set consists of published spreadsheets, it is not surprising that there are not any more unit errors. In set B, the unit inference system detected errors in 7 sheets. A total of 19 instances of unit errors were detected in set B.

It is important to note that a generally accepted "representative" set of spreadsheets

is not available. For the evaluation of our system, we consider spreadsheets in set A "expert" spreadsheets since they have been published with a book, whereas spreadsheets in set B are considered "novice" spreadsheets since they have been developed by students in an introductory course on programming.

5.4 Think-Aloud Study

As part of the outreach component of the EUSES collaboration [46], courses in spreadsheet safety targeted at high school teachers and students have been conducted. A pilot session was conducted in April, 2005 with two high school students. They were given a 20 minute introduction to the UCheck system. After the introductory presentation, the participants were asked to identify and remove unit errors from a spreadsheet seeded with 8 unit errors. It was observed that the two participants successfully removed all 8 unit errors in under five minutes. During the task, it was also observed that the participants debugged the spreadsheet left-to-right and top-to-bottom. This strategy might have served them well since the spreadsheet was *well designed* according to Teo and Tan [115]; input/data cells are to the left and upper part of the spreadsheet and formula cells are to the right and bottom part of the spreadsheet. We also observed that the students were not really using the written spreadsheet specification to help them correct the unit errors. They were simply looking at the cells shaded orange by UCheck, comparing their formulas with the formulas of the neighboring cells, and then changing the formulas of the orange cells. This observation prompted us to design one of the tasks for the later think-aloud study such that this strategy would not work. We were

hoping that would give us better insight into other debugging strategies end users might come up with, especially in those cases where neighboring cells are not available for help/reference.

A course on spreadsheet safety and the use of spreadsheets in Mathematics education was held in July, 2005. The course was spread out over 4 weeks, with 3-hour sessions on 3 days, each week. The first two weeks of the course were targeted entirely at high school teachers. In the class consisting of 10 high school teachers, experience with spreadsheets ranged from 0 to 12 years. The participants were taught how to create spreadsheets using Excel. The discussions and the tasks centered around the use of spreadsheets as a tool for teaching mathematics at the high school level. Quite a bit of the course content was aimed at emphasizing that spreadsheets are prone to a variety of errors. In this context, the participants were given a brief introduction to the WYSIWYT testing methodology and UCheck.

During the sessions with UCheck, the subjects expressed their appreciation for its automatic error-checking capabilities. The subjects also found the fault-localization feature of UCheck a big time saver. When asked about their understanding of units, one of the teachers pointed out that UCheck is more related to the idea of sets (for example, the set of fruits with the set of apples and the set of oranges as subsets) rather than to the physical concept of units (for example, m/s^2 as the unit for speed). This comment was interesting since it illustrates how the teacher was trying to understand UCheck within the framework of mathematical concepts he was familiar with.

In the second week of the course, the participants were split into five groups of pairs. The groups worked with UCheck and WYSIWYT and created their own spreadsheets

and seeded them with errors. These spreadsheets were then used to developed lesson plans. Each group presented the lesson plan they had developed to the other groups to get feedback on the suitability of using the them to teach their high school classes. The goal here was to gauge if the participants had a sufficient understanding of the systems to be able to use them in their classrooms.

In the third week, the teachers used the lesson plans they had developed in the second week to teach twelve high school students how to develop reliable spreadsheets. In the fourth week, the teachers went over a few more exercises with Excel, UCheck, and WYSIWYT.

The workshops showed that high school teachers found UCheck easy enough to understand that they could use it effectively. Moreover, the teachers were able to develop lesson plans and use those to guide high school students on how to create safe spreadsheets. To obtain a better picture of the usability of UCheck, we carried out a think-aloud study, which we will describe in the remainder of this section.

5.4.1 Participants

Five teachers (we refer to them as Subject A through Subject E), from among the ten who were in our course on spreadsheet safety, participated in the think-aloud study. We also used a high school student working with us under the Saturday Academy Program as another subject (Subject F). All the subjects had been introduced to UCheck prior to the study. The study was designed to gauge their understanding of UCheck, and how effective they would be in debugging unit errors.

Subjects A through E had little or no prior programming experience. Subject F was familiar with two programming languages he had used for class projects at high school level. Subjects B and C had never used spreadsheets before our course on spreadsheet safety. All the others had some exposure to spreadsheets (mostly ones they had developed for their own use) prior to the study.

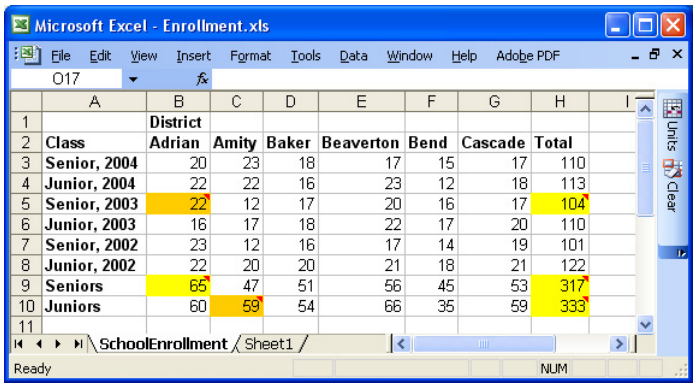
5.4.2 Procedure

At the start of the think-aloud session, the subjects were given a tutorial on UCheck followed by an exercise on how to think aloud. The subjects were then asked to debug unit errors in two spreadsheets.

	A	B	C	D	E	F	G	H
1			Courses					
2		Students	Math	Social Studies	Natural Science	Literature	Student Average	Student GPA
3		M. Abrams	95	82	89	91	89.25	3.57
4		J. Besseck	84	64	69	67	71	2.84
5		T. Cooper	36	87	73	99	73.75	2.95
6		A. Dunn	78	76	85	87	81.5	3.26
7		L. Fannon	82	69	99	76	81.5	3.26
8		J. Greene	69	99	94	72	83.5	3.34
9		I. Harmon	75	87	76	85	80.75	3.23
10		L. Irving	86	63	81	83	78.25	3.13
11		T. James	98	72	75	94	84.75	3.39
12		K. Kady	92	85	88	72	84.25	3.37
13		R. Lennon	76	81	96	67	80	3.20
14		E. McGovern	55	99	92	72	79.5	3.18
15		W. Nash	97	91	100	94	95.5	3.82
16		O. Peterson	88	95	84	94	90.25	3.61
17		Y. Richards	92	72	65	83	78	3.12
18		P. Stone	90	86	42	89	76.75	3.07
19		C. Torok	73	71	67	72	73.75	2.95
20		S. VanBuren	79	68	98	79	81	3.24
21		I. Walters	88	78	72	94	83	3.32
22		P. Young	82	94	88	91	88.75	3.55
23		Class Average	80.75	81.3	81.65	83.05		
24		Class GPA	3.23	3.252	3.266	3.322		

Figure 5.9: Grade sheet with unit errors.

In the first task, the subjects were asked to debug unit errors seeded in the grade sheet



	A	B	C	D	E	F	G	H
1		District						
2	Class	Adrian	Amity	Baker	Beaverton	Bend	Cascade	Total
3	Senior, 2004	20	23	18	17	15	17	110
4	Junior, 2004	22	22	16	23	12	18	113
5	Senior, 2003	22	12	17	20	16	17	104
6	Junior, 2003	16	17	18	22	17	20	110
7	Senior, 2002	23	12	16	17	14	19	101
8	Junior, 2002	22	20	20	21	18	21	122
9	Seniors	65	47	51	56	45	53	317
10	Juniors	60	59	54	66	35	59	333
11								

Figure 5.10: Enrollment sheet with unit errors.

shown in Figure 5.9. Even though this spreadsheet has more cells than the one used in the second task, the errors were more straightforward to debug. Therefore, this task can be considered the easier of the two tasks. The grade sheet was seeded with four unit errors. We seeded the spreadsheets with instances of a few different kinds of unit errors. These errors led to invalid units being inferred in eight other cells. On running UCheck on the spreadsheet, the subjects would initially see the fault-localization feedback shown in Figure 5.9.

In the second task, the subjects were asked to debug unit errors in the spreadsheet shown in Figure 5.10. The spreadsheet was seeded with four errors. The fault-localization feedback presented to the subjects after the first run of UCheck is shown in Figure 5.10. These errors led to invalid units being inferred in four other cells. Only six errors are displayed because cells H9 and H10 each contain two errors. They are shaded yellow in the figure since they have propagation errors from B5 and C10 respectively, in addition to the primary errors we seeded in the cells themselves.

In addition to the audio recording of the think-aloud sessions, we also captured video data of the participants' on-screen interactions. Both sources of data were used for further analysis.

At the end of the two tasks, the subjects were asked to fill out a post-test questionnaire aimed at finding out more about their experiences debugging the spreadsheets using UCheck. The questionnaire also sought feedback on aspects of the system that the subjects would like changed. To gauge the subjects' understanding of the different kinds of unit errors, we asked the subjects to describe in their own words the different types of errors that UCheck would help them detect.

5.4.3 Research Questions

The think-aloud study was aimed at answering the following research questions.

RQ1: *Do end users understand the concept of units well enough to be able to correct the unit errors reported by UCheck? Do they introduce new errors while debugging unit errors?*

UCheck automatically detects unit errors within the spreadsheet and shades the cells that have unit errors. The changes to the cell formulas to correct the unit errors have to be figured out by the user. This effort requires at least a basic understanding of units and the causes of unit errors. Furthermore, RQ1 is important because the system might report unit errors incorrectly in some cases. For example, UCheck relies on automatic header inference to assign units to the core cells. The spatial analysis algorithms of the header inference system work well in practice. Even so, they are not infallible, that is, the

system might report unit errors incorrectly if the headers have been inferred incorrectly. It is therefore important for the user to have a reasonably clear understanding of the requirements of the spreadsheet and the limitations of unit checking so that they can override the system when it is working incorrectly rather than blindly make changes to the spreadsheet formulas.

RQ2: *Does the fault-localization mechanism of UCheck help the end users employ debugging efforts effectively?*

When the user runs UCheck on a spreadsheet, cells that have unit errors are shaded orange, and cells that refer to other cells that have unit errors are shaded yellow. The feedback is aimed at directing the user's attention to the cells that are shaded orange. We were curious if the users would find this feature useful and limit their debugging efforts to the cells that are the sites of unit errors.

RQ3: *What debugging strategies do end users adopt?*

We were also interested in any common strategies or techniques adopted by users while debugging unit errors in spreadsheets since such patterns of behavior would help us refine the system. Improvements or modifications to the system could be targeted at supporting preferred strategies of the users.

5.4.4 Observations

While debugging the unit errors pointed to by UCheck, all the subjects traversed the spreadsheets top-to-bottom. Two of the subjects verbalized their traversal strategies while getting started with the first task at F8.

> "I guess I'll start with F8 since it is the highest cell and work my way down." (Subject F)

> "... an orange one here. I'm just going to start here because that is the first one." (Subject D)

As discussed earlier, the fault localization mechanism of UCheck shades orange the cells that are sites of unit errors. The system shades yellow cells that use cells that have unit errors. The user would have to correct the cells that are shaded orange to make the shading go away—cells shaded yellow might lose their shading once the corresponding source of error has been corrected. Cells H9 and H10 in the enrollment spreadsheet (shown in Figure 5.10) are initially shaded yellow because of the propagated errors. Once the sources of those errors have been corrected, the system identifies the primary unit errors in these cells and shades them orange. Therefore, only in such cases a cell's shading might go from yellow to orange. We looked at ways to analyze the unit error in a cell to determine if it is the result of propagation errors only or the combination of primary and propagation errors. Such analyses would allow us to shade all cells with primary unit error orange, even in those cases in which the cells have propagation errors as well. We found out that it is not trivial to determine this difference by the available unit information. We were concerned that this aspect of the fault localization mechanism might seem counter intuitive to the subjects when they correct the errors in a cell that was shaded orange and the total number of cells shaded orange remains the same or goes up. However, none of the subjects remarked about this effect.

We noticed the following strategies in how the subjects sought feedback from the system about effectiveness of the changes they had made to the cells marked orange by UCheck.

1. *Stepwise debugging* approach: The subject inspects (just) one cell shaded orange, makes a change to the formula and then clicks the "Units" button to seek confirmation the error has been corrected before moving on to the next cell with primary unit error.
2. *Batch debugging* approach: The subject goes through two or more (or potentially all) of the cells that have been shaded orange, makes changes to the formulas, and then clicks the "Units" button to check if the changes done have corrected the unit errors in the spreadsheet.

Subjects B, C, D, and F followed the stepwise debugging approach for both the tasks, whereas Subject A used the batch debugging approach for both the tasks. Subject E switched between the two approaches. During the first task he used stepwise debugging and then used batch debugging for the second task. When asked about it, none of the subjects had any explanation for why they adopted a particular approach over the other in the given situation. However, during the first task Subject C said:

> "Now I think I understand why we'd want to click Units each time. Some of these other ones might go away." (Subject C)

We do not see any advantage of one approach over the other in the case of the tasks used in this study since the formulas are not particularly complex. In the case of spreadsheets with more complex formulas that might cause the user to introduce more errors while

editing formulas, the stepwise debugging approach might work better since it yields immediate rewards and helps the user make steady progress.

We can see from the videos of the sessions and the transcripts that the fault localization mechanism of UCheck guided the subjects' debugging efforts. Three out of the six subjects reviewed the formulas in the cells shaded yellow during the tutorial. None of the subjects spent any time reviewing the error messages or formulas in the yellow cells while working on the tasks. The video data tell us that the fault localization mechanism helped the subjects focus their attention on cells with primary unit errors and not waste their time and effort inspecting other cells. This observation would take on greater significance in the case of larger spreadsheets. Moreover, studies using subjects working with other programming environments have shown that programmers spent an average of 35% of their time navigating between dependencies, and an average of 46% of their time inspecting task-irrelevant code [70]. The fault localization in UCheck helps to avoid these problems to a large degree. The feedback we gathered using the post-session questionnaire also indicates that the subjects found the fault localization mechanism of UCheck very helpful. Five out of the six subjects rated the fault localization mechanism of UCheck as the most useful feature.

> "Coding by orange and yellow helps me find mistakes quickly and lets me check that my edits are correct." (Subject D)

We observed the following strategies adopted by the subjects in correcting the formulas within the cells with unit errors.

Subjects A, B, C, and F used the cells in the neighborhood of the orange cell as examples to guide the changes they made to the formula in the orange cells.

> "... which is an assumption I should check by scrolling down to correct cells ... " (Subject A)

> "... before I do that, I'm sure that's probably right, I'm just going to look up here, at another cell up above ... and I looked above it to double check ..." (Subject C)

Subject D inspected the cells in the neighborhood of orange cells and also looked at the UCheck error message that pops up when the user places the mouse cursor over the shaded cells. The subject would then come up with a prediction of what the correct formula in the cell should be and write it down before clicking on the cell to inspect the formula within the cell and compare with the predicted formula. The subject basically changed the formula in the cell to the predicted formula to correct the errors.

> "I think I am going to look at different student average just to see ... so I am thinking whatever this one is, this orange one, should be an average something[2]-16 ... so this should say AVERAGE(C19:F19) and if it doesn't say that, I'm going to say this is bad. It says AVERAGE(C18:F19), and that's the error. So I am going to change that, C18 to C19, which should be right."

Subject E carried out auditing of the cell values to determine errors within the cells that were shaded orange. This strategy was effective especially in cases in which the cause for the unit errors was not readily obvious as in H9 and H10 from the enrollment sheet.

[2]The subject was trying to figure out the column the cell belongs to.

> "Adding down we get 110, 114, and 315, ... ok, so there's an error somewhere in the adding down or adding across range."

After cross checking the totals in H3, H5, and H7 the subject realized that the error could be in row 9.

> "Well, actually these totals are right. So it should be a horizontal error."

Subject F managed to debug all the unit errors in the two spreadsheets except the ones in H9 and H10. Faced with the orange shading of cells H9 and H10, the subject inspected all the formulas in column H and then changed the formula in cell H9 to SUM(A9:G9). When that did not correct the unit error, the subject went over all the formulas again and decided the formulas were correct. The subject then went over the specifications one more time, took a closer look at the formula in H9 (SUM(B9:G9)), and decided the same result can be obtained by changing the formula to SUM(H3,H5,H7). The subject went ahead and did the change but did not notice that the resulting value in the cell was different from what it used to be. On running UCheck, the subject noticed that the shading in H9 went away. This observation prompted the subject to change the formula in H10 to SUM(H4,H6,H8). These two changes done by the subject resulted in a spreadsheet that is unit correct but still had the formula errors in E9 and F10.

5.4.5 Discussion

We seeded a total of 8 unit errors in the two spreadsheets. Subjects D and E corrected all the errors in the spreadsheet, and Subjects A, B, and C corrected all the errors except

those in cells H9 and H10 in the enrollment spreadsheet. On being told "The error might not necessarily be in the cell that it is being reported in", these three subjects were able to locate and correct the remaining errors as well. As described earlier, Subject F changed the formulas in H9 and H10 such that the unit errors in those cells were corrected but the "errors" originally seeded in E9 and F10 remained uncorrected.[3]

When debugging cells marked as sites of unit errors, five out of six subjects used the neighboring cells that were free from unit errors as examples to guide their debugging efforts. This kind of behavior ("reuse of uses") has been previously documented in other environments [99]. This approach is more common when the users do not really understand the requirements of the spreadsheets they are debugging. Note that the subjects who relied on this technique exclusively (A, B, C, and F) failed to correct the unit errors reported in H9 and H10 without help since the formulas in these cells agree with those in the neighboring cells.

In the case of cells H9 and H10, the cells in their spatial proximity are not in their conceptual proximity. The subjects try to derive conceptual proximity from spatial proximity, and this approach fails in the presence of non-trivial spatial patterns the user is unaware of. This observation indicates that an additional feedback that UCheck could provide is to highlight conceptually similar areas. This strategy could guide the user much better in finding the appropriate "neighbor" cells.

[3] As discussed earlier in the paper, the formulas in cells E9 and F10 by themselves are unit correct.

5.4.6 Results

We present the research questions again and a summary of our findings from the think-aloud study.

RQ1: *Do end users understand the concept of units well enough to be able to correct the unit errors reported by UCheck? Do they introduce new errors while debugging unit errors?*

All participants in the think-aloud study were successful at debugging the unit errors in the spreadsheets except in the cases in which the system reported the errors away from the site of the errors. The subjects did not introduce any new errors while debugging their spreadsheets. The teachers also created their own spreadsheets, seeded with unit errors, and prepared lesson plans to teach their class about unit checking with UCheck. These two aspects indicate that the users developed a working knowledge of units from the tutorials and introductory sessions. Another important observation was that the users could use the system without any knowledge of the underlying formal rule system for unit inference.

RQ2: *Does the fault localization mechanism of UCheck help the end users employ debugging efforts effectively?*

On analyzing the video data collected during the think-aloud study, we observed that the debugging efforts of the participants was guided by the fault localization mechanism of the system. During the pre-session tutorial three out of the six subjects inspected the cells shaded yellow, but all six subjects only inspected the cells shaded orange while performing the study tasks. Since we told the subjects that the cells were seeded only

with unit errors, the subjects indicated they were done debugging the spreadsheets when no cells were shaded orange by the system.

RQ3: *What debugging strategies do end users adopt?*

The think-aloud sessions showed us that the subjects navigate the spreadsheets left-to-right and top-to-bottom. The subjects also used cells in the spatial vicinity of the erroneous cell as examples before making changes to their formula. Four subjects consistently followed the stepwise debugging approach, which gave them immediate feedback about their progress. One subject used the batch debugging approach and made corrections to formulas of all the cells marked orange before invoking the unit checker to seek confirmation that the changes made were indeed correct. One subject switched between the two strategies.

From the participants' comments in the post-session questionnaire, we see that they found UCheck useful. The participants' success at debugging the unit errors show that the system was easy to use.

> "Without UCheck, I probably wouldn't have noticed the more unobtrusive errors in the spreadsheets." (Subject F)

> "Useful for catching errors without having to *calculate*[4] the entire spreadsheet." (Subject C)

[4]On being asked what he meant by "calculate" the subject clarified that he was happy he did not have to step through and check the formulas manually.

5.5 Conclusions and Future Work

In this chapter we described an automatic consistency checker that we have developed for spreadsheets. The main highlights of the system are:

- UCheck carries out consistency checking using units built from the headers that are automatically inferred.
- This type system uses end-user vocabulary for consistency checking, and the headers can also be exploited for generating more user-friendly error messages.

To verify how effective this approach is in practice, we used UCheck in a workshop on spreadsheet safety aimed at high school teachers and students. We discovered that the teachers found the system easy enough to understand and use. Our experiences with courses on spreadsheet safety have shown that teachers and students are able to follow how UCheck exploits header information to carry out consistency checking. The participants of the courses were able to create their own examples of unit errors and explain why they were unit errors. They were also able to identify false positives in their own spreadsheets when they occur as the result of insufficient header information. Moreover, the teachers were able to develop lesson plans and use UCheck to teach high and middle school students how to create safe spreadsheets.

We also conducted a think-aloud study to get a better feel of how well the end users could use the system to debug spreadsheets. The think-aloud study we carried out showed that the participants were able to use the fault-localization feedback provided by UCheck and debug unit errors. The study helped us to identify problems participants had while debugging non-local unit errors. We are currently working on improvements

to the fault-localization mechanism. We also plan to extend the unit inference so that the system can generate suggested changes to correct unit errors.

Chapter 6 – Automatic Test-Case Generation[1]

"I made a mistake", says Memory. "That's impossible", replies Pride. Eventually Memory relents.

Friedrich Nietzsche

Professional software engineers know that programs are likely to contain faults. That is why they invest a considerable amount of time and effort to develop and maintain test suites to detect the faults within their programs. Unfortunately, in spite of the high occurrence of errors within spreadsheets, end users do not have the expertise or tool support to carry out systematic testing of their spreadsheets. In the absence of support for testing in commercially available spreadsheet systems like Microsoft Excel, end users typically test in an ad hoc manner, if at all. Since end users often lack the expertise to come up with effective test cases, they might carry out rudimentary checking of their spreadsheets with historical data (for example, values from previous versions of the sheet). In particular, the lack of test-related feedback leaves users without information about how thoroughly their spreadsheets have been tested, and they come away with an unjustified high level of confidence about their correctness [91].

In addition to monitoring the execution and output of their test cases, end users are also faced with the more complicated task of inventing effective test cases. Even if a testing framework like WYSIWYT were to give the user feedback about the level of

[1]The contents of this chapter are based on [4].

testedness of their spreadsheets, it is not easy to come up with test cases that cause the test suite to reach close to 100% test adequacy. This is where an automatic tool for generating test cases comes into play: The user only has to inspect the output for a generated test case and approve or reject it. Generation and monitoring of test cases would be automatically handled by the system.

In this chapter, we describe an automatic test-case generation system (called AutoTest) that we have developed for spreadsheets. In the next section, we describe a scenario in which an end user is testing a spreadsheet without tool support and how the situation would be different if AutoTest was employed during testing. The test-case generation mechanism of AutoTest targets the definition-use (du) test adequacy criterion. In Section 6.2 we describe how AutoTest identifies du pairs and generates test cases to test feasible du pairs. To evaluate AutoTest, we compared it against the the "Help Me Test" (HMT) test-case generation mechanism in the WYSIWYT testing framework. The evaluation and its results are described in Section 6.3. We present conclusions and directions for future work in Section 6.4.

6.1 Testing With AutoTest

Consider the following scenario in which a user working with the spreadsheet shown in Figure 6.1. The spreadsheet keeps track of the expenses related to office supplies. The amount in B1 (2000 in this case) is the budget allowed for the purchase of office supplies. Rows 4, 5, and 6 store information about the different items that need to be purchased. B4 has the number of pens that need to be ordered, C4 has the cost per pen,

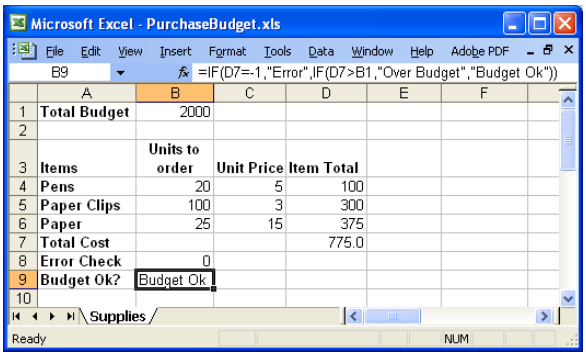


Figure 6.1: Office supplies spreadsheet

and the formula in D4 computes the product of the numbers in B4 and C4 to calculate the proposed expenditure on the purchase of pens. Similarly, rows 5 and 6 keep track of the proposed expenses for paper clips and paper, respectively. The formula in B8 checks to ensure that none of the numbers in B4, B5, or B6 is less than zero. If one or more of the numbers are less than 0, the cell output is 1 to flag the error. Otherwise, the cell output is 0. Cell D7 contains the formula IF(B8=1,-1,D4+D5+D6), which computes the total cost across the three items if the error flag in B8 is not 1. If the error flag in B8 is set to 1, the formula results in -1. The formula in B9 checks if the total proposed expenditure is within the maximum allowed budget for office supplies.

After creating the spreadsheet, the user might inspect the formula cells and try to ensure that the formulas are correct. The user might then use historical data from the previous month as input to verify if the spreadsheet output matches the actual expenses incurred. Once this sort of basic validation has been carried out, the user might be quite confident in the correctness of the spreadsheet and start using it for planning the office expenses. More importantly, overconfidence in the correctness of the spreadsheet might

also keep the user from carrying out the cursory "testing" the next time it is modified.

From a software engineering perspective, the single test case used would not qualify as adequate testing of the spreadsheet. Even if historical data for a few more months were used in testing, it might not give additional coverage since the inputs might only cause the execution of the same parts of the program. Given the absence of support for testing in Microsoft Excel, the user would have to come up with new test cases on their own. Moreover, with no feedback regarding any test adequacy criteria, the user would also not know if any progress was being made as far as adequate testing of the spreadsheet is concerned.

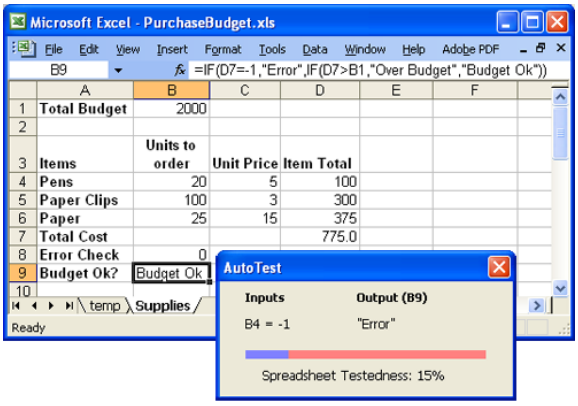


Figure 6.2: Automatically generated test cases for B9

In order to generate test cases for a cell's formula using AutoTest, the user can simply right click in the cell and select the option "Test formula" from the popup menu. Assuming the user chooses to test the formula in cell B9 of the spreadsheet shown in Figure 6.1, the system generates the set of *candidate test cases* for the formula in the cell and presents them as shown in Figure 6.2. A candidate test case is defined as the set of

inputs generated by the system, together with the corresponding output computed by the formula that is to be tested. AutoTest allows the user to perform any one of the following three actions in response to a candidate test case displayed on the user interface.

1. A user can *validate* generated test cases, thereby indicating that the computed output matches the expected output for the formula given the generated input values. Once a user validates a test case, it can be moved to the test suite, and it is displayed on the interface in a green-colored font.
2. A user can *flag* generated test cases to indicate that the computed output value is incorrect given the generated inputs. This action implies that a failure has been detected in the cell. A flagged test case is displayed in red-colored font on the interface, and the cell with the formula that is being tested is also shaded red. The user can inspect the formula within the cell and make changes to correct it since testing detected a failure. Once the formula has been modified, the user can revisit the corrected test case to ensure the computed output matches the expected output for the cell and then validate the test case.
3. A user can also ignore generated test cases if she is unable to decide if the computed output is right or wrong. The user can come back to them at any later point during the course of testing.

For every test case the user approves, the progress bar shows how well tested the spreadsheet program is. Note that, formally, the actual test case is the set of address-value pairs that satisfy the constraints for a du pair. Only the generated values are shown in the AutoTest interface—the values for the other input cells that affect the formula

output that are already in the spreadsheet are implicitly part of the test case and are not shown in the interface.

Internally, the system uses du adequacy to compute the level of testedness. AutoTest saves the user the effort of creating test cases by automatically generating test cases aimed at achieving 100% du adequacy. The automatic generation of effective test suites lowers the cost of testing by reducing and directing the effort invested by the user. Moreover, the progress bar is an accurate indicator of the testedness of the spreadsheet and lets the user know when the spreadsheet has been thoroughly tested.

6.2 Definition-Use Coverage

In the case of du coverage, we can observe that in the case of a single spreadsheet cell, different paths that require different test cases principally result only from IF expressions in that cell's formula. In general, through nested IF expressions, each cell gives rise to a tree of subexpressions that need different test cases to be executed. The leaves of the tree are the possible definitions for the formula in the cell. Uses of definitions of other cells, however, can occur through cell references in the internal nodes or leaves of a cell's formula tree. The method that underlies AutoTest's test-case generation mechanism is based on representing expressions as trees in which internal nodes carry conditions of IF expressions, and leaves of the tree carry arbitrary expressions. The edges of the tree are labeled T or F leading to the subexpressions of the "then" and "else" branches.

6.2.1 Constraint Trees

A formula can be represented as an *expression tree* which stores conditionals in internal nodes and conditional-free subexpressions in the leaves. We can construct an expression tree representation for a formula through two simple transformations. First, we lift all conditionals out of subformulas (that are not conditionals) so that the formula has the form of a nested conditional. This transformation can be performed by repeatedly applying the following semantics-preserving rewrite rule to the conditionals that are subformulas.

$$\psi(\ldots,\mathsf{IF}(c,f_1,f_2),\ldots) \rightsquigarrow \mathsf{IF}(c,\psi(\ldots,f_1,\ldots),\psi(\ldots,f_2,\ldots))$$

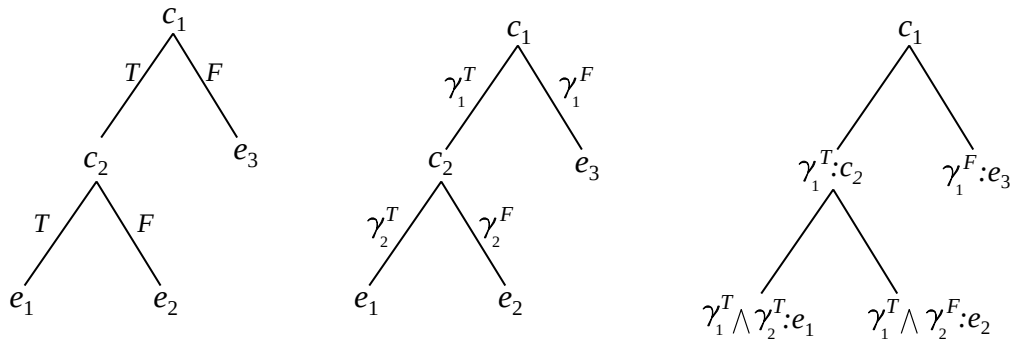
Note that the rewrite rule is only applied when $\psi \neq \mathsf{IF}$.

In a second step, we transform a lifted formula into its corresponding expression tree (see also Figure 6.3(a)) using the function $\mathcal{T}$, which creates for each conditional an internal node labeled with the condition and two subtrees for the two branches. The edges to the branches are labeled T and F to indicate which subtree corresponds to the "then" and "else" branch of the conditional.

$$\begin{array}{lcl} & & \quad c \\ & & T \diagup \quad \diagdown F \\ \mathcal{T}(\mathsf{IF}(c,f_1,f_2)) & = & \mathcal{T}(f_1) \quad \mathcal{T}(f_2) \\ \mathcal{T}(f) & = & f \end{array}$$

The second case leaves all non-conditional formulas unchanged.

Each condition in the internal nodes in an expression tree can potentially evaluate to



(a) Expression Tree (b) Factored Constraint Tree (c) Constraint Tree

Figure 6.3: Stages of test-case generation

true or false. The condition c can therefore be transformed into two constraints γ^T and γ^F that describe the conditions for the execution of the true and false branch, respectively. The constraints have the following form

$$\begin{aligned} \gamma &::= f\,\omega\,v \mid \gamma \wedge \gamma \mid \gamma \vee \gamma \\ \omega &::= < \mid \leq \mid = \mid \geq \mid > \end{aligned}$$

The constraints can replace the edge labels in an expression tree to generate a *factored constraint tree* (see Figure 6.3(b)). For example, a condition $\mathsf{B3} > 4$ will be transformed into the two constraints $\mathsf{B3} > 4$ and $\mathsf{B3} \leq 4$, which will replace the labels T and F, respectively, in the expression tree. The edge labels on a factored constraint tree along each path from the root node to condition c in an internal node or an expression e in a leaf characterize the constraints under which the original formula would evaluate c and e, respectively.

In a final transformation[2] we can collect all the constraints along each path and attach the resulting conditions to the internal nodes and leaves to give rise to a *constraint tree* (see Figure 6.3(c)). For an internal node or a leaf to be executed, the constraints attached to it have to be satisfied. For example, for expression e_1 to be executed, we need both the constraints γ_1^T and γ_2^T to be satisfied. That is why the leaf for e_1 has been annotated with $\gamma_1^T \wedge \gamma_2^T$ in Figure 6.3(c).

6.2.2 Definition-Use Pairs

As mentioned earlier, only a cell whose formula has conditionals or references to other cells that have conditionals can have multiple definitions. Moreover, since a (sub)formula defining a cell may refer to other cells defined by conditionals, a single path is not sufficient to describe a definition. Uses can occur both in internal nodes and leaves. Therefore, uses of a cell C_1 in the formula in another cell C_2 are given by paths from the root of the formula in C_2 to internal nodes or leaves that have references to C_1.

These observations lead to the following concepts relevant to testing du pairs in spreadsheets. Let $\mathcal{C}(a)$ be the constraint tree obtained from the expression tree $\mathcal{T}(S(a))$ as described above. We define the *uses* of a as the set $U_S(a)$, which contains the nodes of all trees $\mathcal{C}(a')$ for which $a \in \sigma(S(a'))$.[3] Correspondingly, the *immediate definitions* of a are given by the the leaves of $\mathcal{C}(a)$. We refer to this set as $D_S^0(a)$. To obtain the complete set of definitions for a we have to combine each expression e in $D_S^0(a)$ with all definitions

[2]Both traversals can be combined into one in the implementation.

[3]The function σ, defined in Section 2.1, takes a spreadsheet formula as input and computes the list of addresses of the cells it references.

for any cell referenced by e, which leads to the following inductive definition for $D_S(a)$, the set of *definitions* of a. $D_S(a)$ is initially defined to be $\{\{\gamma{:}e\} \mid \gamma{:}e \in D_S^0(a)\}$. Then we repeatedly replace a set of paths $P = \{\gamma_1{:}e_1, \ldots, \gamma_k{:}e_k\} \in D_S(a)$ for which $a' \in \sigma(e_i)$ by the set $P \times D_S^0(a')$ until no such a' exists anymore.

Now the set of all du pairs for address a is given by $\{(a,d,u) \mid d \in D_S(a) \wedge u \in U_S(a)\}$. For each du pair we can try to generate a test by solving the constraints stored in the paths (as described in the next section). Whenever the constraint solving fails, a test cannot be generated and an *infeasible* du pair has been identified. A test suite that consists of a test for every feasible du pair is said to be *du-pair adequate*.

6.2.3 Testing Definition-Use Pairs

The generation of a test case for a du pair $(a, \{\gamma_1{:}e_1, \ldots, \gamma_k{:}e_k\}, \gamma_{k+1}{:}e_{k+1})$ requires solving the constraint $\gamma_1 \wedge \ldots \wedge \gamma_k \wedge \gamma_{k+1}$. In a first step, we group the constraints by involved addresses so that we obtain a constraint of the form

$$\gamma^{a_1} \wedge \gamma^{a_2} \wedge \ldots \wedge \gamma^{a_n}$$

where each γ^{a_i} is of the form

$$\gamma_1^{a_i} \vee \gamma_2^{a_i} \vee \ldots \vee \gamma_{k_i}^{a_i}$$

and each $\gamma_j^{a_i}$ is of the form

$$a_i \ \omega \ v_i^1 \wedge a_i \ \omega \ v_i^2 \wedge \ldots \wedge a_i \ \omega \ v_i^{m_{ij}}$$

That is, for each address a_i we obtain an alternative of constraints, each of which determines through a conjunction of value comparisons possible input values for the cell at address a_i. Note that by construction each $\gamma_j^{a_i}$ contains at most one address, namely a_i.

Trying to solve each constraint γ^{a_i} can have one of two possible outcomes.

1. The constraint solver might succeed, in which case, the solution is, for each address, a range of values that directly satisfy the constraints. For each address, any value from the range can be used as a test input to exercise the du pair.
2. The constraint solver might fail. This situation arises when for at least one a_i none of the alternative constraints $\gamma_j^{a_i}$ is solvable. In this case it is not possible to generate any test case that would be able to execute the path. Therefore, failure of the constraint-solving process indicates that the particular path for a definition or use cannot be exercised.

If all constraints have been successfully solved, a test case can be created by taking values from computed ranges for each address and by evaluating the formula to be tested (of which e_{k+1} is a subformula) using these values.[4] A test case has the following form.

$$(\{(a_1, v_1), \ldots, (a_n, v_n)\}, v)$$

[4]The notion of test cases in spreadsheets has been formally defined in Chapter 2.

where $\{(a_1, v_1), \ldots, (a_n, v_n)\}$ are the inputs of the test case, and v is the expected (correct) output.

If the constraint solver fails while trying to solve the constraints for a du pair, that du association cannot be exercised given the constraints. In other words, unsolvable constraints on input data cells allow us to automatically detect infeasible du pairs in the spreadsheet program. In general, it might not be possible to execute all of the du associations in spreadsheets. The problem of identifying infeasible du pairs in programs written in general-purpose programming languages is undecidable [120]. Detection of infeasible du pairs is decidable in the case of spreadsheets languages like Excel since they do not have loop constructs or recursion.

To illustrate how AutoTest works, consider a spreadsheet composed of the following three cells.

A1: 10

A2: IF(A1>15,20,30)

A3: IF(A2>25,A2+10,A2+20)

Cell A1 is an input cell and therefore has one definition and does not use any other cells. A2 has one use of A1 (which is always executed) in the condition. Since the formula in A2 has two branches, it has two definitions with the constraints $\gamma_1^T \equiv \mathsf{A1} > 15$ and $\gamma_1^F \equiv \mathsf{A1} \leq 15$ for the true and false branches, respectively.

The formula in A3 has three uses of A2, one in the condition, and one in each of the true and false branches. The use in the condition is always executed, but we need to satisfy the constraints $\gamma_2^T \equiv \mathsf{A2} > 25$ and $\gamma_2^F \equiv \mathsf{A2} \leq 25$ in order to execute the true and false branches, respectively. Therefore, to test the formula in A3 to satisfy the du-

adequacy criterion, we need test cases that execute the two definitions of A2 and the two uses of A2 in the branches of A3's formula—a total of 4 du pairs. The constraints for the 4 du pairs are shown below.

$$\{\gamma_1^T\text{:20}, \gamma_2^T\text{:A2+10}\}, \{\gamma_1^T\text{:20}, \gamma_2^F\text{:A2+20}\}$$
$$\{\gamma_1^F\text{:30}, \gamma_2^T\text{:A2+10}\}, \{\gamma_1^F\text{:30}, \gamma_2^F\text{:A2+20}\}$$

We consider the constraints individually in the following.

- $\{\gamma_1^T$:20, γ_2^T:A2+10$\}$: γ_2^T requires A2 to be some value greater than 25. However, satisfying γ_1^T results in 20 in A2. Therefore, the two constraints cannot be satisfied at the same time and we have an infeasible du pair.

- $\{\gamma_1^T$:20, γ_2^F:A2+20$\}$: Satisfying γ_1^T results in 20 as the output of A2 which satisfies the constraint γ_2^F. Solution of γ_1^T results in A1 $= 16$ as the test case to execute this du pair.

- $\{\gamma_1^F$:30, γ_2^T:A2+10$\}$: Satisfying γ_1^F results in an output of 30 in A2. This value satisfies the constraint γ_2^T. Therefore, the test case that exercises this du pair is A1 $= 14$, which is obtained by solving γ_2^T.

- $\{\gamma_1^F$:30, γ_2^F:A2+20$\}$: Satisfying γ_1^F results in 30 in A2. This value violates the constraint γ_2^F. Therefore, this set of constraints cannot be solved and we have an infeasible du pair.

Even though the solution of a constraint might result in many test cases, AutoTest generates only one test case per feasible du pair. For example, any value greater than 15

in A1 would satisfy γ_1^T, but AutoTest generates 16 as the test case. Moreover, in a more complicated spreadsheet, there might be one or more independent sets of constraints (representing different paths between a du pair) for a feasible du pair. In such cases, each solvable set of constraints could generate one or more test cases. Once again, AutoTest only generates one test case per feasible du pair. For each test case that is generated, the user would have to verify if the computed output is correct for the given inputs. The decision was made to generate only one test per feasible du pair to minimize the effort required of the user.

6.3 Evaluation

For a test-case generator for spreadsheets to be useful, it has to be both *effective* and *efficient*. Effectiveness is judged with respect to a test-adequacy criterion, du adequacy in this case. A more effective tool in this respect would be one which is capable of generating test cases that exercises more of the feasible du pairs. In Section 8.3, we describe the evaluation of the test suites generated by AutoTest using mutation testing. Efficiency is measured in terms of the time taken by the system to generate the test cases that meet the adequacy criterion. This factor is especially important in the case of spreadsheet systems with their support for immediate visual feedback.

For the evaluation, we compared AutoTest against HMT, which is the only other automatic test case generator available for spreadsheets. We followed the evaluation of HMT described in [50] and took into consideration two dependent variables.

1. Ultimate effectiveness, defined as the percentage of the total number of feasible

du associations exercised by the generated test cases

2. Response time for test generation

Since HMT uses randomization, depending on the technique used, the measures of the dependent variables may change from one run to another. Therefore the ultimate effectiveness score for HMT is averaged over 35 runs and the median effectiveness and response time scores over 35 runs have been presented in [50]. Our system, on the other hand, always produces the same output given the same starting spreadsheet configuration. We reproduce the results from [50] in Tables 6.1 and 6.2 and compare with the numbers obtained by running AutoTest on the same spreadsheets. Note that the response time figures for HMT (shown in Table 6.2), which were taken from [50] are from 1999.

Spreadsheet	HMT				AutoTest
	Random (without ranges)	Random (with ranges)	Chaining (without ranges)	Chaining (with ranges)	
Budget	99.6%	100.0%	100.0%	100%	100%
Digits	59.4%	97.9%	100.0%	100%	100%
FitMachine	50.5%	50.5%	97.9%	97.9%	100%
Grades	67.1%	99.8%	99.7%	99.9%	100%
MBTI	25.6%	100.0%	99.9%	99.6%	100%
MicroGen	71.4%	99.2%	100.0%	100%	100%
NetPay	40.0%	100.0%	100.0%	100%	100%
NewClock	57.1%	100.0%	99.0%	99.4%	100%
RandomJury	78.8%	83.2%	94.3%	92.7%	100%
Solution	57.7%	78.8%	100.0%	100%	100%

Table 6.1: Ultimate effectiveness scores of the different techniques per spreadsheet.

AutoTest is also efficient in the sense that it only generates one test case per feasible du pair for the formula that is being tested. Therefore it generates the minimum number of test cases to be able to execute all feasible du pairs. Such optimal test suites can also

Spreadsheet	HMT				AutoTest
	Random (without ranges)	Random (with ranges)	Chaining (without ranges)	Chaining (with ranges)	
Budget	3.8	3.9	9.9	10.6	< 0.01
Digits	2.2	10.1	34.5	28.4	< 0.01
FitMachine	3.8	3.9	13.5	14.3	< 0.01
Grades	7.7	18.6	14.2	14.4	0.5
MBTI	37.5	40.0	31.2	30.7	1.0
MicroGen	2.7	8.1	6.4	6.5	< 0.01
NetPay	1.2	1.2	1.7	1.7	< 0.01
NewClock	2.3	2.5	10.4	9.3	< 0.01
RandomJury	80.2	28.7	182.3	173.2	2.0
Solution	1.3	1.4	18.9	17.9	< 0.01

Table 6.2: Response times (in seconds) of the different techniques per spreadsheet.

save the user time and effort during creation of test suites (since the user has to approve a candidate test case before it can be added to the suite of test cases for the formula), test runs, and maintenance of test suites. The ultimate effectiveness scores reported in Table 6.1 and the response times reported in Table 6.2 for AutoTest are based on the optimal test suites that are generated by the system. Information about the size of the test suites generated by HMT for achieving the level of coverage shown in Table 6.1 is not available.

As can be seen from the ultimate effectiveness scores shown in Table 6.1, AutoTest generates test cases that cause the execution of all feasible du associations for the spreadsheets used in the evaluation. HMT running the Chaining algorithm has comparable ultimate effectiveness scores. From the response time figures shown in Table 6.2, we also see that AutoTest is very fast in generating test cases. Based on these results, we can conclude that AutoTest, whose approach to generate test cases is based on derivation, propagation, and solving of constraints, is efficient and accurate. Moreover, AutoTest can also detect infeasible du associations automatically

6.4 Conclusions and Future Work

In this chapter, we have described AutoTest, an automatic test-case generator that supports users of Microsoft Excel in the systematic testing of their spreadsheets. AutoTest automatically generates a minimal set of tests for each formula cell that guarantees a du adequate test coverage. The system runs efficiently and also produces, as a by-product, information about infeasible du associations in the spreadsheet.

The test-case generation approach, which is based on constraint generation, propagation, and solving, is conceptually simple, which is important for at least two reasons. First, it allows its reuse in other systems. For example, we believe that it would be straightforward to integrate this new technique into the WYSIWYT tool. Second, it facilitates extensions to be investigated in future. For example, we plan to extend AutoTest to allow whole regions to be tested by the test cases for a single region-representative formula. We can base this extension effectively on the region inference mechanisms reported in [5]. In future work, we also plan to evaluate the scalability of this approach to large spreadsheets.

Detecting faults is only the first step in correcting a spreadsheet. Fixing incorrect formulas is generally required to remove faults. Therefore, the integration of AutoTest with a testing framework like WYSIWYT and a debugger like GoalDebug (described in Chapter 7) is very important to provide end users with a comprehensive solution to detect and correct faults within their spreadsheets.

Chapter 7 – Debugging End-User Spreadsheets[1]

Would that I discover truth as easily as I can uncover falsehood.

Cicero

Programmers spend a major portion of their time identifying and correcting faults within their programs. A recent study conducted in the U.S. by the National Institute of Standards and Technology (NIST) has found that software engineers typically spend 70-80% of their time testing and debugging. On average, errors take 17.4 hours to find and fix [114]. The problem is even more serious in the case of end users developing software, thereby making the case for the need for innovative tool support for end-user debugging. Any approach to end-user debugging faces two challenges:

1. *Fault localization:* Even in cases where end users identify a failure within their spreadsheet, the fault(s) that led to the failure might not be obvious. The debugger should help the end user isolate the causes of the failure.
2. *Program changes:* Once a fault has been identified by the user, the program needs to be changed to correct it. It has been observed in empirical studies that even when end users identify faults, they have trouble with the formula syntax and introduce more errors while correcting them [94].

Data collected during studies to evaluate the effectiveness of fault localization techniques in WYSIWYT have shown that end users make mistakes while editing formulas.

[1]The contents of this chapter are based on [2, 9].

	Gradebook	Payroll
Number of subjects	51	51
Total errors	154	381
Errors on values	144	168
Errors on formulas	10	213
Total formula-edit errors	454	361
Correct to incorrect	81	68
Incorrect to incorrect	373	293

Table 7.1: User mistakes during debugging

It was observed that subjects make many wrong decisions (so called *oracle mistakes*) while testing their spreadsheet formulas [94]. The number of instances of oracle mistakes are shown in Table 7.1. In particular, we see that there were 373 instances in the Gradebook task and 293 instances in the Payroll task when the subjects pinpointed the faults correctly, but then went on to make incorrect changes to the cell formulas. We see from the table that the vast majority of formula-edit errors fall into this category.

We have developed a system, called GoalDebug, that allows the users to specify the expected output for any cell whose output is incorrect. The system then uses static and dynamic slicing techniques to generate change suggestions, any one of which when applied, would result in the expected output being computed in the cell. This approach addresses the two challenges pertaining to fault localization and program changes discussed above as follows.

1. The generated change suggestions are ranked (from most likely to least likely) by a set of heuristics. The change-inference mechanism and ranking heuristics help isolate the fault(s).
2. The change suggestions that get generated can be directly applied to the spreadsheet formulas, thereby minimizing the need for any editing of the formula by the

end user.

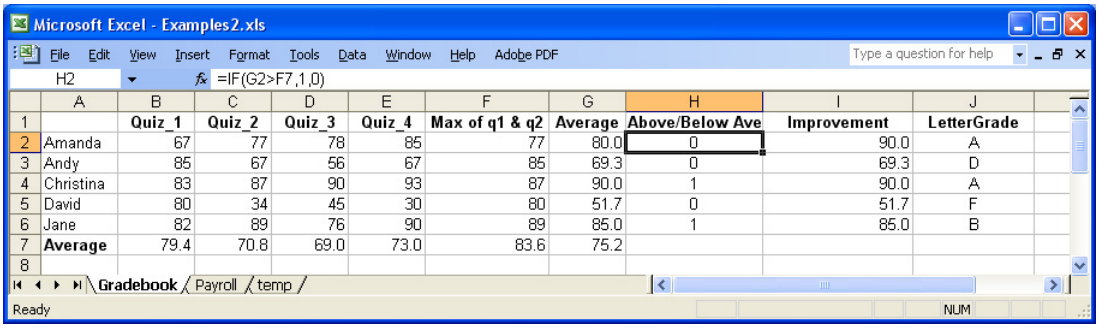
Typically, debugging involves finding out how the inputs have been incorrectly used to compute the incorrect results. GoalDebug inverts the "debugging question": It determines the program changes that would result in the expected output being computed.

In the next section, we describe how a user might interact with GoalDebug. In Section 7.2, we describe how the system infers the change suggestions based on the constraints specified by the user. A principal difficulty of the GoalDebug approach is that a different output in a cell can, in general, be achieved through many different changes in that or other cells that impact it. We employ heuristics to rank the generated change suggestions so that the correct one shows up as high as possible in the ranked list of suggestions that get presented to the user. The heuristics used to rank the change suggestions are described in Section 7.3. We carried out an empirical evaluation to assess the effectiveness of GoalDebug. The evaluation and its results are described in Section 8.4. We present conclusions and directions for future research in Section 7.4.

7.1 Debugging With Change Suggestions

Consider the spreadsheet shown in Figure 7.1 used to store the grades of the students in a course. The overall grade scored by a student is determined from the student's scores on 4 quizzes (stored in columns B through E). The instructor allows the student to drop the lower score between Quiz_1 and Quiz_2. The average score for each student (and the class) is computed by the formulas in column G. Column H checks to see if a student's average score is "above average" (designated by 1) or "below average" (designated by

0) when compared to the class average, computed in cell G7.



	A	B	C	D	E	F	G	H	I	J
1		Quiz_1	Quiz_2	Quiz_3	Quiz_4	Max of q1 & q2	Average	Above/Below Ave	Improvement	LetterGrade
2	Amanda	67	77	78	85	77	80.0	0	90.0	A
3	Andy	85	67	56	67	85	69.3	0	69.3	D
4	Christina	83	87	90	93	87	90.0	1	90.0	A
5	David	80	34	45	30	80	51.7	0	51.7	F
6	Jane	82	89	76	90	89	85.0	1	85.0	B
7	**Average**	79.4	70.8	69.0	73.0	83.6	75.2			
8										

Figure 7.1: Gradebook spreadsheet

In the spreadsheet, Amanda's average score is 80, which is above the class average of 75.2. Therefore, the value in H2 should be 1 instead of 0.[2] On observing this discrepancy in the output values, the user can specify the expected output for H2 using the interface shown in Figure 7.2, which can be invoked by right clicking in the cell and selecting Debug from the drop-down menu options. In this case, the user specifies the expectation that the output of H2 should be equal to 1.

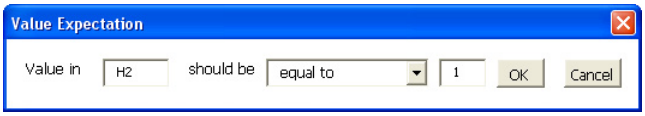


Figure 7.2: Specifying expected output for a cell

Based on the input from the user, the system generates change suggestions and ranks them using the set of heuristics. The cell with the highest ranked change suggestion is

[2]The computed value is incorrect because the formula in cell H2 incorrectly compares Amanda's average score in G2 against the number in F7 instead of G7, as can be seen from the formula bar in Figure 7.1.

shaded orange, and all other cells for which suggestions have been generated are shaded yellow. In this example, suggestions are only generated for cell H2 in the first step. The user can view the generated change suggestions by right clicking in the cell and picking Suggestions from the drop-down menu options. The top five suggestions are displayed as shown in Figure 7.3. The user can ignore or select any change suggestion on the list. The user can also ask the system to generate more change suggestions. We see in Figure 7.3 that the suggestion for changing F7 to G7 is the second one on the list of (the top five) possible changes that would result in the formula in H2 evaluating to 1. This change would correct the fault in the formula.

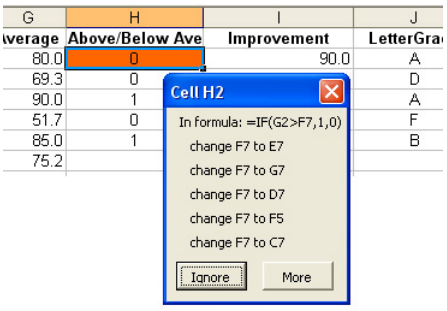


Figure 7.3: Change suggestions

If the user selects this change suggestion, by left clicking on it, she is presented with the confirmation dialog shown in Figure 7.4. This dialog allows the user to make any changes (if necessary) to the suggested modified formula before applying it to the cell. Once the user clicks the Apply button, the change is made to the spreadsheet formula. We see in this example that only three actions (specifying the expected output for the cell with the error, selecting the change suggestion, and applying the change suggestion) were required of the user to correct the error in the formula. The cognitively difficult task of coming up with the correction

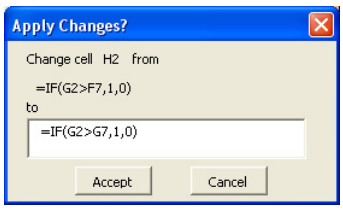


Figure 7.4: Applying suggestion

and entering the new correct formula in the cell (ensuring syntactic correctness) was taken care of by the system.

7.2 Change Inference

As described in the previous section, GoalDebug allows the user to specify the expected value for a cell using the debugging interface shown in Figure 7.2. Given the expected value w for a cell a, GoalDebug converts it into a *target constraint* γ on the cell output. A target constraint has the following form:

$$\gamma \quad ::= \quad \omega v \mid \gamma \wedge \gamma \mid \gamma \vee \gamma \mid \lambda x.\gamma$$

where $\omega \in \{<, \leq, =, \geq, >\}$. In addition to value constraints (ωv), *and* constraints allow the formulation of ranges of values as expectations, and *or* (as well as *and*) constraints capture results of constraint propagation. Lambda abstractions are needed to define constraint transformations. Since a constraint γ defines a value predicate, it can be applied to values, as in $\gamma(v)$. For example, $[< 3 \wedge \geq 1](2)$ yields true.

Given a constraint γ, change inference computes a set of change suggestions expressed in terms of γ, which can later be converted to a value. A suggestion has the form $a : f \rightsquigarrow \gamma$ to express that the (sub)formula that is contained in the cell a should be changed such that it evaluates to a value v for which $\gamma(v)$ is true. The inference of change suggestions is formalized through the function δ shown in Figure 7.5. In the implementation, the cases are tried in the given order (decreasing specificity). In the

$$\delta(a,v,\gamma) = \{a : v \rightsquigarrow \gamma\} \quad (1)$$

$$\delta(a,\mathsf{AND}(e_1,\ldots,e_k),\gamma) = \{a : \mathsf{AND} \rightsquigarrow \mathsf{OR} \mid [\![\mathsf{OR}(e_1,\ldots,e_k)]\!]_S = v \wedge \gamma(v)\} \quad (2a)$$

$$\delta(a,\mathsf{OR}(e_1,\ldots,e_k),\gamma) = \{a : \mathsf{OR} \rightsquigarrow \mathsf{AND} \mid [\![\mathsf{AND}(e_1,\ldots,e_k)]\!]_S = v \wedge \gamma(v)\} \quad (2b)$$

$$\delta(a,f(e_1,...,c,...,e_k),\gamma) = \{a : c \rightsquigarrow \uparrow a' \mid [\![f(e_1,...,\uparrow a',...e_k)]\!]_S = v \wedge \gamma(v)\} \quad (3)$$

$$\delta(a,f(\uparrow a_1,...,\uparrow a_n),\gamma) = \{a : f(\uparrow a_1,...,\uparrow a_n) \rightsquigarrow f(\uparrow a_1,...,\uparrow a_n,\uparrow a') \mid \rho(S,a,\uparrow a') \wedge [\![f(\uparrow a_1,...,\uparrow a_n,\uparrow a')]\!]_S = v \wedge \gamma(v)\} \quad (4a)$$

$$\delta(a,f(\uparrow a_1,...,\uparrow a_n),\gamma) = \{a : f(\uparrow a_1,...,\uparrow a',...,\uparrow a_n) \rightsquigarrow f(\uparrow a_1,...,\uparrow a_n) \mid \uparrow a' \in \{\uparrow a_1,...,\uparrow a_n\} \wedge [\![f(\uparrow a_1,...,\uparrow a_n)]\!]_S = v \wedge \gamma(v)\} \quad (4b)$$

$$\delta(a,f(e_1,\ldots,e_k),\gamma) = \cup_{i=1}^{k}\delta(a,e_i,f^i(e^i)(\gamma)) \cup \{a : f(e_1,\ldots,e_k) \rightsquigarrow \gamma\} \cup \{a : f \rightsquigarrow f' \mid [\![f'(e_1,\ldots,e_k)]\!]_S = v \wedge \gamma(v)\} \quad (5)$$

$$\delta(a,\uparrow a',\gamma) = \delta(a',S(a'),\gamma) \cup \{a : \uparrow a' \rightsquigarrow \uparrow a'' | [\![S(a'')]\!]_S = v \wedge \gamma(v)\} \cup \{a : f(e_1,\ldots,e_k) \rightsquigarrow \gamma\} \quad (6)$$

$$\delta(a,\mathsf{IF}(p,e,e'),\gamma) = \begin{cases} \delta(a,e,\gamma) & \text{if } [\![p]\!]_S = T \wedge [\![e']\!]_S = v \wedge \neg\gamma(v) \quad (7a) \\ \delta(a,e',\gamma) & \text{if } [\![p]\!]_S = F \wedge [\![e]\!]_S = v \wedge \neg\gamma(v) \quad (7b) \\ \delta(a,p,=F) \cup \delta(a,e,\gamma) & \text{if } [\![p]\!]_S = T \wedge [\![e']\!]_S = v \wedge \gamma(v) \quad (7c) \\ \delta(a,p,=T) \cup \delta(a,e',\gamma) & \text{if } [\![p]\!]_S = F \wedge [\![e]\!]_S = v \wedge \gamma(v) \quad (7d) \end{cases}$$

$$\delta(a,e_1\ r\ e_2,\gamma) = \{a : r \rightsquigarrow r' \mid [\![e_1\ r'\ e_2]\!]_S = v \wedge \gamma(v) \wedge r' \in (\{<,\leq,>,\geq,=,\neq\} - \{r\})\} \quad (8)$$

Figure 7.5: Change Inference

following discussion, we explain each of the definitions in Figure 7.5 with examples.

If a cell with address a contains a constant, say v, but the target constraint is γ (with $\neg\gamma(v)$), the suggestion generated by definition (1) is to change the constant v to another constant w (*target value*) for which $\gamma(w)$ holds. How the target value w is generated from the constraint γ is explained later.

Empirical evidence suggests that end users often confuse logical connectors [88]. That is, they use OR when they mean AND and vice versa. Subject to satisfaction of the constraints on the cell, definition (2a) recommends replacing AND with an OR connector, and definition (2b) recommends replacing OR with an AND connector. Along similar lines, definition (8) recommends changing relational operators in conditions.

A constant in a formula might be replaced with another constant that satisfies the constraints. This suggestion would be generated by definition (1). However, the constant could also be replaced with references to cells that have values that satisfy the constraints as captured by definition (3).

In the case of aggregation formulas that operate over a range of cells, definition (4a) generates suggestions that include additional references in the range, and definition (4b) generates suggestions that recommend removing references from the range. When recommending additional references to include, we need the additional check $\rho(S, a, \uparrow a')$ to ensure that the inclusion of reference $\uparrow a'$ in the formula in a does not introduce a cyclic reference in the spreadsheet S.

If a constraint γ is applied to a formula $f(e_1, \ldots, e_k)$ which evaluates to v (given $\neg\gamma(v)$), there are three possibilities for deriving a change suggestion. The first option is to change the formula itself so that it evaluates to value w for which $\gamma(w)$ holds. The second option is to try to "backpropagate" the constraint γ to the arguments e_i in the formula. How this is done in practice depends on the operation f. In general, we need k constraint transformations $f^1, \ldots, f^k$ that can compute the change required for any argument that causes the formula $f(e_1, \ldots, e_k)$ to evaluate to a value that satisfies γ. We abbreviate the sequence $e_1, ..., e_{i-1}, e_{i+1}, ..., e_k$ by e^i and write $f^i(e^i)(\gamma)$ to refer to the constraint for the ith argument of f. This constraint must be defined to satisfy the following implication.

$$f^i(e^i)(\gamma) = \gamma' \Longrightarrow \qquad (\forall v.\gamma'(v) \Longrightarrow \gamma(f(e_1, ..., e_{i-1}, v, e_{i+1}, ..., e_k))$$

For example, the constraint transformations for $+$ are defined as follows.

$$+^1(v_2)(\gamma) = \lambda x.\gamma(x - v_2)$$
$$+^2(v_1)(\gamma) = \lambda x.\gamma(x - v_1)$$

To see how this works, consider a spreadsheet S composed of the following three cells:

$$\{(\mathsf{A1},3),(\mathsf{A2},5),(\mathsf{A3},\mathsf{A1}+\mathsf{A2})\}$$

Assume that the output of A3 is required to satisfy the constraint > 11 even though $[\![\mathsf{A1}+\mathsf{A2}]\!]_S = 8$. In this case, $+^1$ derives for cell A1 the constraint $\lambda x.[> 11](x-5)$, which can be simplified to $\lambda x.[> 6](x)$ and further to (> 6). Similarly, $+^2$ derives for cell A2 the constraint $\lambda x.[> 11](x-3) = \lambda x.[> 8](x) = (> 8)$. Both constraints can be converted by the function $\mathcal{V}$ (shown below) into values (here, integer values 7 and 9, respectively). Therefore, by applying either suggestion we obtain $[\![\mathsf{A1}+\mathsf{A2}]\!]_S = 12$ which satisfies the condition > 11 on the output of A3. Therefore, for a function of k arguments, we can propagate the constraints to (and potentially generate suggestions from) each of the k arguments. The third option is to generate change suggestions that replace the function f with others that would result in the output value v such that $\gamma(v)$. All three possibilities have been combined in definition (5).

For a cell reference, changes are inferred for the referenced cell, and the address can be changed to any other cell a'' that evaluates to a value that satisfies γ. Moreover, the reference itself can be replaced by a constant that satisfies γ, see definition (6).

In the case of conditional formulas, we provide more detailed change inference since we have more detailed information about the data flow from subformulas than in the generic case of definition (5). We distinguish four cases depending on the result of the predicate and on whether or not one of the alternatives evaluates to a value satisfying γ. If cell a contains the formula $f = \mathsf{IF}(p,e,e')$ and the condition p evaluates to true, f

evaluates to its first alternative. That is, $[\![f]\!]_S = v$ where $[\![e]\!]_S = v$ with $\neg\gamma(v)$. Therefore, reasonable change suggestions can be obtained through $\delta(a, e, \gamma)$. This case is captured in definition (7a). If e' evaluates to w with $\gamma(w)$, any change that causes p to evaluate to false is also a reasonable change, see definition (7c). The two other cases, (7b) and (7d), are obtained by an analogous consideration of p evaluating to false and e evaluating to w with $\gamma(w)$.

We can observe that the function δ propagates constraints through formulas while the system reports values in the user interface. Once δ has propagated the initial constraint into a set of change suggestions, which still contain constraints, these change constraints are converted into values by a function $\mathcal{V}$. First, the constraint to be converted is simplified as much as possible, for example, $< 3 \wedge \leq 1$ can be simplified to ≤ 1. After that $\mathcal{V}$ can produce value suggestions for constraints that do not contain $\wedge$ or $\vee$.

$$\begin{aligned}
\mathcal{V}(\omega v) &= v \text{ for } \omega \in \{\leq, =, \geq\} \\
\mathcal{V}(< v) &= \max_T\{w \mid w < v\} \\
\mathcal{V}(> v) &= \min_T\{w \mid w > v\} \\
\mathcal{V}(\gamma) &= \gamma
\end{aligned}$$

The functions $\max_T$ and $\min_T$ are type-dependent maximum and minimum functions. For example, $\max_T\{w \mid w < 3\}$ yields 2 if w is an integer, while it yields 2.99 if w is a floating point value.[3] In cases when γ is a non-simple constraint that cannot be solved, the user is presented with a suggestion that is a textual description of the constraint itself.

[3]The number of decimal places is arbitrarily fixed to 2.

7.3 Ranking Heuristics

In general, the system generates many change suggestions for any given value expectation. It is unrealistic to expect the user to inspect the complete list of generated suggestions to identify the correct one to apply. Therefore we use a set of heuristics to rank the generated suggestions and use cell shading to direct the user's attention to the cells with the highest ranked change suggestions.

Similarities in spreadsheet formulas have been exploited in consistency checking [80] and in testing spreadsheets [31]. The idea of *node equivalence classes* described in Section 2.3 gives a measure of similarities between formulas, and we use it for ranking change suggestions for formulas. For example, copy equivalence all the cases of the formula in a cell being created by copying and pasting the formula from another cell. On the other hand, two formulas are considered to be structurally equivalent if they contain the same operations in the same order. Obviously, copy equivalence is a stronger similarity condition than structural equivalence. Therefore, if a change suggestion recommends changing a formula f to g that is copy equivalent to f, the change suggestion in ranked higher than a suggestion that recommends changing f to a formula h that is only structurally equivalent to f.

In addition to the node equivalence classes, we define *spatial similarity* as the similarity of formulas in spatially co-located cells. In some cases, cells with similar formulas might not be in the immediate spatial neighborhood of each other. This situation might arise when the cells under consideration are fulfilling similar conceptual roles in different regions of the spreadsheet. To express this idea, we define *conceptual similarity* as

the similarity of formulas in cells that are not spatial neighbors.

The heuristics that rank a change suggestion that recommends replacing formula f within a cell a with formula g takes into consideration three components:

1. The similarity between f and g based on the node equivalence class they belong to.
2. The number of formulas in the spreadsheet that are similar to g.
3. The Manhattan distance between a and other cells that contain formulas that are similar to g.

This approach has been adopted to allow the spatial and conceptual neighbors to induce a higher ranking for a formula that is similar to others within the spreadsheet. In addition to the above, formula changes are also ranked using additional information from the following 4 sources.

Unit analysis. As described in Chapter 5, the UCheck system performs automatic consistency checking of spreadsheet formulas and detects unit errors. By integrating UCheck with GoalDebug, we rank unit-correct change suggestions higher than suggestions that are not unit correct. This integration allows us to add a level of consistency checking to the change-suggestion process. Change suggestions are also type checked to ensure that the application of a suggested change would not introduce a type error in the spreadsheet.

Impact analysis. The generated change suggestions are also ranked on the basis of the number of cells that would become affected by applying a particular suggestion. For example, assume that cells c_1, c_2, and c_3 have references to cell c_4, whereas the output

of cell c_5 is only used by c_6. Other things being equal, a change to the formula in c_5 would be preferred over a change to the formula in c_4 since it has lower impact on other cells within the spreadsheet.

User confidence. We assign a level of confidence to the generated change suggestion based on how the user specifies the expected value for a cell. For example, if the expected output for cell c_1 is equal to v_1, and the expected output for cell c_2 is less than v_2, we assume that the user is more confident about the expected outcome for c_1 than for c_2. All other factors remaining the same, the change suggestions generated from the constraint on c_1 would be ranked higher than those generated from the constraint on c_2. Moreover, the confidence levels are used to resolve conflicts when propagated constraints from different sources are encountered.

Data-flow information. An observed failure in a cell a in a spreadsheet could be due to a fault within the cell's formula or due to a fault in a cell upstream from a in the data-flow chain. When more that one cell has been marked incorrect by the user, any common cells upstream from the marked cells are potential sites of faults. Therefore the change suggestions for these cells get ranked high. Moreover, the change suggestions for cells that have been marked as correct can be ranked low—we do not filter these out, just in case the user made a mistake.

Locality. Apart from the considerations that go into ranking changes to entire formulas, changes to references are ranked based on the Manhattan distance between the cells addressed by the original reference and the new one. This approach makes neighboring cells more likely suggestions. This heuristic is based on the assumption that the introduction of incorrect references is primarily the result of mechanical error (clicking

an incorrect cell to select the target).

Type information. Finally, changes to values are ranked based on their types. For example, a suggestion that recommends changing an integer to a float would have a lower rank than one that recommends changing an integer value to another integer value.

An empirical evaluation we carried out of GoalDebug is described in detail in Section 8.4. In the evaluation, we picked a set of spreadsheets that have been used in studies described in [4, 50] and seeded them with errors using a suite of mutation operators we have designed (the operators are described in Chapter 8). The evaluation showed that out of the 3988 mutants that were generated, GoalDebug was unable to correct the mutations in only 117 cases. In other words, GoalDebug was effective at correcting the mutations in 97% of the cases. Moreover, we also found that the top ranked suggestion corrects the mutations in 59% of the cases and the top two suggestions correct the mutations in 71% of the cases. The suggestion that corrected the mutation is ranked within the top five in 80% of the cases.

7.4 Conclusions and Future Work

In this chapter, we have described a new approach to debugging, called "goal-directed debugging", that we have developed to help end users debug spreadsheets. We have described an implementation and how it infers program change suggestions to correct faults. We have also described the heuristics used to rank the generated change suggestions in order to help the user locate the correct change suggestion more easily.

The current version of the user interface of GoalDebug is rather rudimentary since

it does not support user's interaction with the constraints. In future work, we plan to incorporate mechanisms that allow users to view, add, delete, or modify the constraints associated with any cell directly.

The question of whether the fault lies in the cell marked by the user or one that is upstream in the data flow can be resolved by seeking more information from the user by shading the cells that are upstream and asking if their outputs are correct. Once a cell that has a fault has been isolated, change suggestions for it can be ranked the highest. This approach would solicit more feedback from the user and minimize the number of change suggestions generated by first trying to isolate the fault(s). It can also be very effective in cases where multiple faults lead to a common point of failure that is identified and marked by the user.

In future work, we plan to conduct user studies to evaluate the effectiveness (in terms of time taken and number of faults identified and corrected) of end users using GoalDebug to debug spreadsheets. One potential problem with GoalDebug could be that users might trust the system too much and always pick the top-ranked suggestion, even when the top-ranked suggestion is not the right one. The user studies would also give us insights into debugging strategies employed by end users when program changes are suggested to them by the system.

Chapter 8 – Mutation Operators for Spreadsheets[1]

We read the world wrong and say that it deceives us.

Rabindranath Tagore

Mutation analysis is a technique used to measure the quality of test suites [39, 61]. The program that is being tested is seeded with errors to generate *mutants*. Test cases are developed with the goal of detecting as many mutants as possible. In practical terms, if the test cases in the test suites detect all mutants, then the program is well tested by the test suites [85]. The fault seeding is typically done by means of *mutation operators*. Ideally, the mutation operators are *adequate* in the sense that they are able to show how effective the test suite is, and *effective* in terms of testing time.

The error-seeding capabilities of mutation operators can also be used to modify programs for empirical studies of error-detection systems. The advantage of having a collection of operators is that the designer of the study can chose the types of errors that are seeded, depending on the goals of the evaluation.

In the next section, we present a formal overview of mutation testing. In Section 8.2 we describe the collection of mutation operators we have designed for spreadsheets. In Section 8.3 we describe how the mutation operators were used to evaluate the mutation adequacy of the du-adequate test suites generated by AutoTest. We then present two case studies of how the operators were used to evaluate two systems: GoalDebug in Section 8.4, and a combination of WYSIWYT and UCheck in Section 8.5.

[1] The contents of this chapter are based on [4, 9, 10, 73].

8.1 Mutation Testing

Let P be a program that has the input domain D and codomain D'. Running P on an input $x \in D$ is written as $P(x)$. We also use $P(x)$ to refer to the result of the program run, that is, $P(x) \in D'$. A *test case* is a pair $(x,y) \in D \times D'$. A program *passes* a test $t = (x,y)$ if $P(x) = y$, otherwise P *fails* the test t. A *test set* is a set of tests $T = \{t_1, \ldots, t_n\}$. A program *passes* a test set T if P passes every $t \in T$, otherwise P *fails* the test set T.

Assume we make n copies of P and introduce a single mutation in each one to get the mutated versions P_1 through P_n. (Mutants generated by introducing a single mutation in the original program are called first-order mutants. Higher-order mutants are generated by introducing two or more mutations in the original program.) Let $T \subset D \times D'$ be a passing test set, that is, P passes or satisfies every test in T. To test the *mutation adequacy* of the test set T, we run it against each of the mutants. We say a mutant is *killed* when it fails T, whereas mutants that pass T are said to be *alive*. The basic assumption is that if T kills a mutant, then it will detect real unknown faults as well. Extending the idea, if T kills all the *non-equivalent* mutants, it would be capable of detecting a wide variety of unknown faults as well. A mutant P_i is said to be equivalent to P if $\forall x \in D, P(x) = P_i(x)$, and this equivalence is expressed as $P_i \equiv P$. Obviously, an equivalent mutant cannot be killed by testing. *Mutation adequacy* or the *mutation score* is computed as

$$\frac{\text{number of killed mutants}}{\text{total number of non-equivalent mutants}} \times 100\%$$

In choosing the mutation operators, we make the following assumptions laid out in [27].

1. The *competent programmer hypothesis*: Given a functional specification SPEC, the programmer is competent enough to produce a program P that is within the immediate "neighborhood" of the program P^* that satisfies SPEC. Any program that is far removed from the neighborhood of P^* is said to be *pathological*. The hypothesis allows us to limit the programs we need to consider by excluding the pathological ones.

2. The *coupling effect*: Complex faults within a program are linked to simple faults in such a way that a test suite that detects all simple faults within a program will also detect most complex faults.

It has been shown that if the program is not too large, only a very small proportion of higher-order mutants survives a test set that kills all the first-order mutants [63, 64, 82]. This result, which supports the coupling effect, is helpful for cases in which a single failure is the manifestation of multiple faults. Therefore, if we have tests that detect the isolated faults, we would be able to detect a compound fault (which is a combination of two or more simple faults detected by the test suite) with a high level of probability.

In the following sections, generation of mutant f_i from formula f using mutation operator o is shown as $f \overset{o}{\rightarrowtail} f_i$. Note that f (and f_i) can be a subformula of a larger formula.

8.2 Mutation Operators for Spreadsheets

Mutation operators are designed to satisfy one or both of the following criteria:

1. They should introduce syntactical changes within the program that replicate errors typically made by programmers.
2. They should force a tester (working towards killing non-equivalent mutants) to develop test suites that achieve standard test adequacy criteria like statement and decision coverage.

To satisfy the first criterion, we have designed mutation operators that reflect errors reported in the spreadsheet literature [17,47,91]. In order to satisfy the second criterion, we have included operators that have been developed for general-purpose programming languages.

A "standard set" of 22 mutation operators have been proposed for FORTRAN in [83]. Not all of these can be adapted for spreadsheets. The subset of operators that are applicable to spreadsheets is shown in Table 8.1. It has also been shown empirically in [83] that test suites that kill mutants generated by the 5 operators ABS, AOR, LCR, ROR, and UOI are almost 100% mutation adequate when compared to the full set of 22 operators. This result suggests that we could potentially lower the cost of mutation testing by using the 5 operators instead of the full set of 22 operators.

Some adjustments have to be made in adapting mutation operators for general-purpose languages to the spreadsheet domain. The correspondence of the constructs in the two domains is shown in Table 8.2. Since we do not have any means of distinguishing between data cells that are equivalent to input data from the data cells that correspond to constants within a program written in a general-purpose programming language, we treat all data cells within a spreadsheet as inputs to the spreadsheet program.

Operator	Description
ABS	*ABS*olute value insertion
AOR	*A*rithmetic *O*perator *R*eplacement
CRP	*C*onstants *R*e*P*lacement
CSR	*C*onstants for *S*calar variable *R*eplacement
LCR	*L*ogical *C*onnector *R*eplacement
ROR	*R*elational *O*perator *R*eplacement
SCR	*S*calar for *C*onstant *R*eplacement
SDL	*S*tatement *D*e*L*etion
SRC	*S*ou*R*ce *C*onstant replacement
SVR	*S*calar *V*ariable *R*eplacement
UOI	*U*nary *O*perator *I*nsertion

Table 8.1: Subset of mutation operators for FORTRAN

General-purpose language	Spreadsheets
Input data	Data cells
Constants	Data cells
Variables	Cell references
Statement	Cell formula
Output data	Output of formula cells

Table 8.2: Correspondence of constructs in general-purpose programming languages and in spreadsheets

The mutation operators we have devised for spreadsheets are shown in Table 8.3. In cases where they have been adapted from the operators shown in Table 8.1, the original operators are shown in parenthesis. In the description of the operators, the use of "constant" (in the operators CRP, CRR, and RCR) refers to constants within spreadsheet formulas. We have also included a few additional operators that only make sense in the context of spreadsheets. For example, we have introduced a set of operators that mutate ranges, replace formulas in cells with constants, and change functions used in formulas (these operators are discussed in more detail below).

Operator	Description
ABS	*ABS*olute value insertion
AOR	*A*rithmetic *O*perator *R*eplacement
CRP	*C*onstants *RP*lacement
CRR	*C*onstants for *R*eference *R*eplacement (adapted from CSR)
LCR	*L*ogical *C*onnector *R*eplacement
ROR	*R*elational *O*perator *R*eplacement
RCR	*R*eference for *C*onstant *R*eplacement (adapted from SCR)
FDL	*F*ormula *DeL*etion (adapted from SDL)
FRC	*F*ormula *R*eplacement with *C*onstant
RFR	*ReF*erence *R*eplacement (adapted from SVR)
UOI	*U*nary *O*perator *I*nsertion
CRS	*C*ontiguous *R*ange *S*hrinking
NRS	*N*on-contiguous *R*ange *S*hrinking
CRE	*C*ontiguous *R*ange *E*xpansion
NRE	*N*on-contiguous *R*ange *E*xpansion
RRR	*R*ange *R*eference *R*eplacement
FFR	*F*ormula *F*unction *R*eplacement

Table 8.3: Mutation operators for spreadsheets

As shown in Table 8.2, we consider references in spreadsheets equivalent to variables in general-purpose programming languages. Therefore, the equivalent of a variable replacement in a general-purpose programming language would be a change in reference in spreadsheets. Mutation operators for references should ensure that they do not introduce cyclic references in spreadsheets. Cyclic references lead to non-terminating computations and are automatically detected by spreadsheet systems like Excel. Therefore, there is no need to kill such mutants using test cases. In the following, we discuss three approaches to refine mutations of references in the context of the competent programmer hypothesis and the error detection capabilities of spreadsheet systems.

1. It is reasonable to assume that changes to a reference are only to a cell of the same type. For example, a reference to a cell that has a numeric value should

only be changed to a reference to a cell that has a numeric value. On the other hand, depending on the application, it might make sense in some cases to remove this restriction on types since spreadsheet systems like Excel do not perform any systematic type checking. As a result, spreadsheet programmers might introduce such errors in their spreadsheets and not know about it.

2. In case of formulas that operate over cells within the same row (or column), it is reasonable to change references only to other cells within the same row (or column) to reflect the user's lack of understanding of the specification. This refinement could be too restrictive if we are trying to model mechanical errors in which the user accidentally clicks a cell in the immediate neighborhood of the cell they meant to include in the formula.
3. To model mechanical errors, it is reasonable to change a reference to a cell to other cells in the immediate spatial neighborhood of the original reference. The "distance" between the original cell and the new cell could be considered a measure of the reference mutation, and could be tuned depending on the application. If we are modeling mechanical errors only, it would not be reasonable for the mutated reference to point to a cell far removed from the original one.

The operators can be tuned depending on what they are being used for. For example, having more powerful mutations would result in better test cases. On the other hand, it might make sense to restrict the operators if they are being used to seed errors in spreadsheets for empirical evaluations. In the current implementation of the mutation operators, we impose the third constraint (and drop the second) discussed above since we use the operators to seed mechanical errors in the evaluation of spreadsheet tools

described in Sections 8.3, 8.4, and 8.5. Since Excel does not carry out systematic type checking of spreadsheet formulas, we do not enforce the first constraint during mutation, thereby requiring the user to generate test cases to kill mutants that have type errors not detected by Excel. As we have shown in [6], integrating a type checker with Excel would help prevent type errors and lower the cost of mutation testing by reducing the number of mutants that need to be killed.

We discuss below the mutation operators that we have designed that are unique to the spreadsheet domain.

Range mutation: Aggregation formulas in spreadsheet typically operate over *contiguous* or *non-contiguous* ranges. For example, the formula SUM(A2:A10) computes the sum over the contiguous range from A2 through A10, whereas the formula SUM(A2,A5,A8,A11) computes the sum over the non-contiguous range that includes the references A2, A5, A8, and A11. We employ the following novel operators to mutate ranges.

1. Contiguous Range Shrinking (CRS): This operator shrinks a contiguous range by altering the reference at its beginning or end. For example, SUM(A1:A5) $\overset{\text{CRS}}{\rightarrowtail}$ SUM(A1:A4)
2. Non-contiguous Range Shrinking (NRS): This operator shrinks a non-contiguous range by removing a reference. For example, SUM(A1,A3,A5) $\overset{\text{NRS}}{\rightarrowtail}$ SUM(A1,A5).
3. Contiguous Range Expansion (CRE): This operator expands a contiguous range by altering the reference at its beginning or end. For example, SUM(A1:A5) $\overset{\text{CRE}}{\rightarrowtail}$ SUM(A1:A6).
4. Non-contiguous Range Expansion (NRE): This operator expands a non-

contiguous range by introducing an additional reference. For example, SUM(A1,A3,A5) $\overset{\text{NRE}}{\rightarrowtail}$ SUM(A1,B2,A3,A5).

5. Range Reference Replacement (RRR): This operator can replace a reference within a range. For evaluating the effectiveness of test suites, the RRR operator can be allowed to operate on any reference within a contiguous or non-contiguous range. Note that even in the case of a contiguous range, the RRR operator does not have to be limited to the start and end references of the range. For example, SUM(A1:A4) $\overset{\text{RRR}}{\rightarrowtail}$ SUM(A1,B2,A3,A4). In this case, the operator inspects the contiguous range and then replaces any reference within the range. However, for the purpose of seeding errors for empirical studies, the RRR operator can be restricted to apply to any reference within a non-contiguous range or to the references at the start and end of a non-contiguous range. This restriction is reasonable for this application since it more closely reflects possible user errors.

We treat contiguous and non-contiguous ranges differently because of the competent programmer hypothesis. In general, we only consider mutations to contiguous ranges that affect the first or last reference whereas mutations to non-contiguous ranges can happen to any reference within the range.

Formula Replacement: The FRC (Formula Replacement with Constant) operator overwrites the formula in a cell with the computed output value from the formula. It has been observed in empirical studies that when users do not understand the specifications of the spreadsheet well, they sometimes overwrite the formula cells with the expected output as a "quick fix" to get the output they expect [34]. The FRC operator has been included in the collection of mutation operators to model this kind of error.

Function Mutation: Due to a poor understanding of the requirements or the spreadsheet system, users sometime use incorrect functions in their formulas. For example, a user might use SUM (and forget to divide by COUNT of the cells) instead of using AVERAGE. We include the FFR (Formula Function Replacement) operator to model this error. As in the case of mutation of cell references, we use a *distance* measure for FFR operator. For example, replacement of AVERAGE with SUM (or vice versa) seems more likely than replacing SUM (or AVERAGE) with an IF-statement.

8.3 Mutation Coverage of Du-Adequate Test Suites

The primary goal of mutation testing is to evaluate the quality of test suites. In this section, we describe a study we carried out to evaluate the mutation adequacy of the du-adequate test suites generated by AutoTest. An adequacy criterion like du adequacy is well defined in the sense that it is independent of the programming language or programming environment. The effectiveness of mutation testing, on the other hand, depends on the design of the mutation operators. A poor choice of mutation operators might certify that a test suite is 100% mutation adequate even when a program is poorly tested. The all-uses data-flow criterion for test adequacy was empirically evaluated against mutation adequacy in [84], and the authors of that paper found that mutation-adequate test suites are closer to satisfying the data-flow criterion and succeed at detecting more faults. It has also been shown in the context of general-purpose programming languages that mutation-adequate test suites are guaranteed to have such properties as extremal values [81], covering all statements [32, 33], and all branches [33]. It is not known if these

results hold in the spreadsheet domain.

8.3.1 Experiment Setup

For the purpose of evaluating the mutation adequacy of the du adequate test suites generated by AutoTest, we used spreadsheets that have been used in the empirical evaluations described in [4, 50, 100]. We designed our experiment to answer the following research questions.

RQ1: *What is the mutation adequacy of the automatically generated test suites for each one of the spreadsheets?*

We have shown empirically in [4] that the test suites generated by AutoTest are 100% du adequate. RQ1 compares the effectiveness of du adequacy against mutation adequacy for each one of the spreadsheet used in this evaluation.

RQ2: *What types of mutations are hard to detect? In other words, what operators generate mutants that are hard to kill?*

Identifying the types of mutants that are hard to kill would guide us to types of faults that are hard to detect through testing. If these faults occur infrequently in real-world spreadsheets, there is not much cause for concern. On the other hand, if such faults occur frequently in practice, it would make a strong case in favor of more research into tool support for preventing or identifying them.

RQ3: *How effective are the automatically generated test suites at killing higher-order mutants?*

Because of the coupling effect, a test suite that kills most of the first-order mutants

would also be effective at killing higher-order mutants. In this context, we would like to determine the mutation adequacy of du adequate test suites against higher-order mutants.

8.3.2 Results

As described in Chapter 6, AutoTest generates du adequate test suites using a constraint-based approach. For each test case that is generated, the user would have to make the judgement if the output is correct for the given set of inputs. To minimize the effort required of the user, for each feasible du pair, AutoTest generates only one test case. However, the system is capable of generating more than one test case per feasible du pair in the cases in which more than one independent set of constraints for a du pair is solvable. Therefore, in the evaluation we carried out, we ran AutoTest under three different configurations to study the mutation adequacy of the automatically generated test suites. The data shown in Table 8.4 is for du-adequate test suites in which only one test case (from the first solvable set of constraints) was generated by AutoTest per feasible du pair. We see from the data that the mutation adequacy of these test suites is low in many cases.

We then modified AutoTest to pick a solvable constraint set at random from those available for a du pair. The randomly picked set of constraints was used to generate one test case for each du pair. The mutation adequacy (averaged across 5 runs) of the du-adequate test suites generated using this approach are shown in Table 8.5. We see from the results that this approach generates test suites that have higher mutation coverage than the previous one.

Spreadsheets	Mutants generated	Mutants killed	Mutation coverage
MicroGen	176	53	30.1%
Grades	338	302	89.35%
FitMachine	440	397	90.23%
Digits	465	452	97.2%
NetPay	108	108	100%
PurchaseBudget	325	318	97.85%
RandomJury	886	380	42.89%
Sales	338	338	100%
Solution	235	235	100%
Budget	158	117	74.05%
MBTI	1145	740	64.63%
NewClock	321	260	81%

Table 8.4: Mutation adequacy (running first test per definition-use pair)

In the third configuration, we modified AutoTest to generate test cases for all solvable constraint sets. This approach is expensive from the user's perspective since the number of test cases generated is higher than the two cases discussed above. This approach is also wasteful from a du-adequacy point of view since multiple test case that exercise the same du pair will be generated in this case. The mutation coverage percentages for this configuration are shown in Table 8.6. We see that the mutation adequacy scores are high for the test suites, but the results are not really surprising since the additional coverage is achieved by the large number of test cases that are generated.

From a user-effort point of view, the more coverage (in this case du coverage or mutation coverage) a given test case provides, the better it is. In Table 8.7, we present comparisons between the number of test cases generated and the corresponding mutation coverage for the three configurations of AutoTest described above. "Rand1", "Rand2", and "Rand3" are three different runs in which the sets of constraints are randomly picked from those available for each du pair. "First" picks the first set of constraints that can

Spreadsheets	Mutants generated	Mutants killed	Mutation coverage
MicroGen	176	150	85.2%
Grades	338	302	89.35%
FitMachine	440	416.3	94.61%
Digits	465	452	97.2%
NetPay	108	90	83.33%
PurchaseBudget	325	270.8	83.32%
RandomJury	886	771.3	87.05%
Sales	338	338	100%
Solution	235	235	100%
Budget	158	131.8	83.42%
MBTI	1145	880.5	76.9%
NewClock	321	277	86.29%

Table 8.5: Mutation adequacy (running single randomly selected test per definition-use pair) averaged over 5 runs)

be solved for any du pair. In Table 8.7 there are some cases in which the number of test cases generated by these four approaches differ for the same spreadsheet. This variation is due to those cases in which constraints from multiple du pairs generate the same test case. Even though such a test case would be associated with multiple du pairs that it tests, it is counted only once. "All" generates test cases for all solvable sets of constraints. The data in Table 8.7 show that test suites with more test cases have higher mutation coverage in general. We also observe that some test suites are more effective at killing mutants than others. For example, for the "NewClock" spreadsheet, the 14 test cases picked in "Rand1" were as mutation adequate as the 16 tests picked by "Rand2".

The test suites generated in the "All" configuration are successful at killing almost all the mutants. That is, the test suites that are 100% du adequate, are, in many cases, close to being mutation adequate as well. However, the number of test cases generated in the "All" configuration is considerably higher than the number of test cases generated in

Spreadsheets	Mutants generated	Mutants killed	Mutation coverage
MicroGen	176	175	99.43%
Grades	338	302	89.35%
FitMachine	440	426	96.82%
Digits	465	452	97.2%
NetPay	108	108	100%
PurchaseBudget	325	325	100%
RandomJury	886	886	100%
Sales	338	338	100%
Solution	235	235	100%
Budget	158	158	100%
MBTI	1145	1011	88.3%
NewClock	321	277	86.29%

Table 8.6: Mutation adequacy (running all tests)

the other configurations, thereby making it substantially more expensive. The empirical results of the evaluation confirm that mutation adequacy is a stronger test adequacy criterion than du adequacy. More importantly, especially when adopting the strategy of generating a single test case for every du pair, we need to pick those test cases that help the overall test suite achieve higher mutation adequacy. The strategy of picking the du-adequate test suite with higher mutation adequacy will make the test suites generated by AutoTest more powerful since it draws from two strong test-adequacy criteria.

8.3.3 Analysis of Surviving Mutants

In order to answer RQ2, we analyzed the mutants that survived the test suites since they are obviously difficult to kill. The types of mutants that survived the du-adequate test suites generated under the "All" configuration of AutoTest for each of the spreadsheet are shown in Table 8.8.

Spreadsheets	Rand1		Rand2		Rand3		First		All	
	Tests	Cov.	Tests	Cov.	Tests	Cov.	Tests	Cov.	Tests	Cov.
MicroGen	14	80.7%	15	89.2%	16	99.4%	12	30.1%	28	99.4%
Grades	45	89.3%	44	89.3%	41	89.3%	37	89.3%	298	89.3%
FitMachine	23	96.8%	24	96.8%	22	90.2%	20	90.2%	27	96.8%
Digits	7	97.2%	7	97.2%	7	97.2%	7	97.2%	7	97.2%
NetPay	4	100%	4	100%	4	100%	4	100%	4	100%
PurchaseBudget	17	100%	19	100%	18	100%	17	97.8%	46	100%
RandomJury	37	57.6%	37	100%	36	100%	24	42.9%	522	100%
Sales	1	100%	1	100%	1	100%	1	100%	1	100%
Solution	11	100%	10	100%	11	100%	10	100%	17	100%
Budget	5	74.1%	5	93%	5	93%	5	74.1%	114	100%
MBTI	101	77.7%	100	76.4%	101	78.3%	72	64.6%	834	88.3%
NewClock	14	86.3%	16	86.3%	15	86.3%	11	81%	29	86.3%

Table 8.7: Comparison of number of test cases against mutation coverage

The data in Table 8.8 show that the mutants that survived the du-adequate test suites were generated by 6 operators. In the following, we discuss the operators together with examples of mutants they generate, to show why constraint-based automatic test case generation is not sufficient to kill all non-equivalent mutants. The part of the formulas that have been changed by the mutation have been marked by a box.

Arithmetic operator replacement (AOR): In the FitMachine spreadsheet, cell B7

Spreadsheets	AOR			CRP			CRR			LCR			RFR			ROR		
	A	T	%	A	T	%	A	T	%	A	T	%	A	T	%	A	T	%
Microgen							1	24	4.2									
Grades				4	13	30.8	8	55	14.5				20	224	8.9	4	20	20
FitMachine	3	66	4.5	3	26	11.5	2	65	3.1				5	261	1.9	1	20	5
Digits	2	81	2.5	1	37	2.7							10	258	3.9			
MBTI	25	249	10	10	139	7.2	19	135	14.1	9	96	9.4	71	518	13.7			
NewClock	6	42	14.3	4	24	16.7	6	45	13.3				28	191	14.7			
Total	36	438	8.2	22	239	9.2	36	324	11.1	9	96	9.4	134	1452	9.2	5	40	12.5

Table 8.8: Mutation operators and live mutants (A: #alive mutants, T: Total #mutants seeded,%: Survival percentage)

contains the formula

IF(B8>5,B9/B8*B5 * 0.8,IF(B8<2,B9/B8*B5*1.5,B9/B8*B5))

and the mutant

IF(B8>5,B9/B8*B5 / 0.8,IF(B8<2,B9/B8*B5*1.5,B9/B8*B5))

survives the test suite. The formula in B5 is IF(B3<0,0,B3), and cell B3 contains the value 0, which causes cell B5 to evaluate to 0. Therefore, even though the test suite has a test case that causes the condition B8 $>$ 5 to evaluate to True, B9/B8*B5*0.8 $=$ B9/B8*B5/0.8 $= 0$, since B5 $= 0$. Hence the mutant survives the test suite. Therefore, to be able to kill the mutant, we need the additional constraint that B9 $\neq 0 \wedge$ B5 $\neq 0$. The constraint B5 $\neq 0$ would get propagated backwards and result in the constraint B3 > 0 being imposed on B3.

Constants for reference replacement (CRR): In the FitMachine spreadsheet, cell B10 contains the formula

IF(B5 =0,0,IF(B5<5,B6*B7/2,IF(B5>20,B6*B7*2/3,B6*B7))).

The CRR operator causes the formula to mutate to

IF(5 =0,0,IF(B5<5,B6*B7/2,IF(B5>20,B6*B7*2/3,B6*B7))).

The condition $5 = 0$ evaluates to False. However, since B7 evaluates to 0 (as discussed in the context of the example for AOR above), both the original formula and the mutated formula for the cell B10 always result in 0. Therefore, the mutant cannot be killed by the du-adequate test suite unless we include the additional constraint $B6 \neq 0 \wedge B7 \neq 0$.

Logical connector replacement (LCR): In the Budget spreadsheet, cell B8 contains the formula

$$IF(\boxed{OR}(B4<0,B5<0,B6<0),1,0).$$

This formula is a check to ensure that the quantities of various office supplies ordered are not negative numbers. The LCR operator generates the mutant

$$IF(\boxed{AND}(B4<0,B5<0,B6<0),1,0).$$

In the original spreadsheet (which is considered as a default test case), the values in B4, B5, and B6 are all less than zero. This test case causes both the original and mutated formula to evaluate to True and result in 1 as the output of cell B8. In order to force the condition OR(B4<0,B5<0,B6<0) to evaluate to False, AutoTest generates a test case in which $B4 = B5 = B6 = 0$. However, this test case also causes the mutated condition AND(B4<0,B5<0,B6<0) to evaluate to False, thereby causing the mutant to survive the test suite.

Reference replacement (RFR): In the Grades spreadsheet, cell B15 computes the final grade of a student with the formula

$$IF(B14=1,\text{"Input Error"},IF(\boxed{B13}>89,\text{"A"},IF(B13>79,\text{"B"},$$
$$IF(B13>69,\text{"C"},IF(B13>59,\text{"D"},\text{"F"}))))).$$

The mutated formula

IF(B14=1,"Input Error",IF(B12 >89,"A",IF(B13>79,"B",
IF(B13>69,"C",IF(B13>59,"D","F")))))

survives the tests since the value in B12 and B13 are the same in the absence of extra-credit assignments. Therefore, a du-path adequate test suite for the formula in B15 fails to kill this mutant since it does not force the values in B12 and B13 to be different.

Relational operator replacement (ROR): The ROR operator mutates the formula in B15 in the Grades spreadsheet to

IF(B14=1,"Input Error",IF(B13>89,"A",IF(B13>79,"B",
IF(B13>69,"C",IF(B13 $\geq$ 59,"D","F")))).

The mutant survives the test suite since the test case that causes the original condition B13 $>$ 59 to evaluate to True has the value in B13 as 60. That is, in none of the test cases is the value in B13 exactly equal to 59, and so the outputs from both the original formula and the mutant are the same for all the test cases.

As can be seen from Table 8.8, the reference replace operator (RFR) has the highest impact in terms of the number of mutants that survive testing. This operator was designed to mirror error situations in which users accidentally select the wrong cells while building formulas (mechanical error) or select the wrong cell because they do not understand the specification for the spreadsheet correctly (logical error). No data is available on the frequency of occurrence of this class of errors in real-world spreadsheets. However, it would be very beneficial to extend AutoTest to kill mutants generated by the RFR operator to improve the mutation adequacy of the generated test suites.

Spreadsheets	Mutants generated	Mutants killed	Mutation coverage
MicroGen	7659	7659	100%
Grades	49221	49074	99.70%
FitMachine	76222	73251	96.10%
Digits	87746	87649	99.89%
NetPay	4584	4584	100%
PurchaseBudget	48880	48590	99.4%
RandomJury	373211	373211	100%
Sales	53928	53928	100%
Solution	18505	18505	100%
Budget	9819	9811	99.92%
MBTI	635259	603599	95.02%
NewClock	45193	45108	99.81%

Table 8.9: Mutation adequacy of AutoTest test suites with second-order mutants

8.3.4 Mutation Testing With Higher-Order Mutants

As can be seen from Table 8.9, the number of second-order mutants generated is considerably higher than the number of first-order mutants. This factor, in itself, would make mutation testing with higher-order mutants expensive to the point of being infeasible. To verify the validity of the coupling effect in the spreadsheet domain (RQ3), we investigated the mutation adequacy of the automatically generated test cases with second-order mutants. The coverage numbers for second-order mutants is shown in Table 8.9. Comparing these numbers with those in Table 8.6, we see that the test suites kill about the same percentage of first- and second-order mutants. This result confirms similar results reported for general-purpose programming languages [63, 64, 82]. The observation that test suites that are effective at killing first-order mutants are equally effective at killing second-order mutants as well indicates that we only need to carry out mutation testing using first-order mutants.

8.3.5 Extending AutoTest

Based on the analysis of the mutants that survive the du-adequate test suites, we see that we might be able to extend the system to increase the mutation-adequacy score of the automatically generated test suites by killing more mutants generated by the operators shown in Table 8.8. To kill a mutant f_i generated from a formula f, we need a test case $t = (I, v)$ such that $[\![f]\!]_I = v \wedge [\![f_i]\!]_I \neq v$. In the following, we describe how test cases can be generated to kill mutants by considering which mutation operator was used to create the mutant. In the following discussion, for $f \overset{\text{o}}{\rightarrowtail} f_i$ we implicitly assume that the generated test case should also cause the du path with f (and f_i) to be executed. The goal is to improve the mutation adequacy of the du-adequate test suites generated by AutoTest using additional constraints.

In practice, test-case generation would have to be performed in two steps. In the first step, generate the constraints for executing du pairs within the spreadsheet. In the second step, impose the constraints to refine the test suites in order to target mutation adequacy. If the inclusion of additional constraints prevents the generation of any test cases for a given du pair, ignore the additional constraints. On the other hand, in cases where the inclusion of the additional constraints results in sets of constraints that are solvable, the generated test case would exercise the du pair and also kill more mutants.

Why does test-case generation have to be done in two steps? If the constraints for both du adequacy and mutation adequacy were combined and solved together, the resulting test suite generally would not even be du adequate. The adverse impact is not surprising since the combined set of constraints is more restrictive than either one con-

sidered independently. Therefore, we need to ensure that at the very least, the test cases for exercising the feasible du pairs are included in the automatically generated test suites by adopting the two-phase approach outlined above.

We now analyze the operators from Table 8.8 that have surviving mutants even after all the du adequate tests have been run. We see that in some cases, we can include additional constraints that would kill more mutants. However, for a few of the operators, the generation of these additional constraints is not feasible.

1. *A*rithmetic *O*perator *R*eplacement (AOR): For the arithmetic operators $\{+,-,*,/\}$, given any two subformulas f and g, we have a total of 12 possible cases under AOR. We consider two example to show how extra constraints can be generated to target these mutants.

   - $f+g \overset{\text{AOR}}{\rightarrowtail} f-g$

     To be able to kill this kind of mutant, the generated test case should satisfy the additional constraint $[\![g]\!]_I \neq 0$.

   - $f/g \overset{\text{AOR}}{\rightarrowtail} f*g$

     In this case, the generated test case should satisfy the additional constraint $[\![g]\!]_I \neq 1$.

2. *C*onstants *R*e*P*lacement (CRP): This operator replaces constants within subformulas. For example, we have $c_1 \overset{\text{CRP}}{\rightarrowtail} c_2$ where c_1 and c_2 are constants. If this mutation occurs on constants that are part of an arithmetic operation, a test case that executes the arithmetic expression might kill the mutant. However, in some cases, the mutants generated might be equivalent to the original program. For ex-

ample, if c_1 is the input to the SQUARE function, $c_1 \overset{\text{CRP}}{\rightarrowtail} -c_1$ cannot be killed by testing. The situation is more complex in the case of conditionals. For example, assume $c_1 < c_2$ and the following cases

- $\text{IF}(p < c_1, f, g) \overset{\text{CRP}}{\rightarrowtail} \text{IF}(p < c_2, f, g)$
- $\text{IF}(p < c_2, f, g) \overset{\text{CRP}}{\rightarrowtail} \text{IF}(p < c_1, f, g)$
- $\text{IF}(p > c_1, f, g) \overset{\text{CRP}}{\rightarrowtail} \text{IF}(p > c_2, f, g)$
- $\text{IF}(p > c_2, f, g) \overset{\text{CRP}}{\rightarrowtail} \text{IF}(p > c_1, f, g)$

where p, f and g are sub formulas. To kill the above mutants we need test cases that satisfy the constraint $c_1 < [\![p]\!]_I < c_2$.

3. *C*onstants for *R*eference *R*eplacement (CRR): This operator replaces references with constants. Given a reference a, we have $a \overset{\text{CRR}}{\rightarrowtail} c$, where c is a constant. In this case, we need a test case that satisfies the constraint $[\![a]\!]_I \neq c$ in addition to the constraints for du adequacy. Since the original reference a cannot be inferred by inspecting the mutant formula, there is no direct way to generate the constraint $[\![a]\!]_I \neq c$. The most straightforward solution would be to consider each constant c within a formula and generate test cases such that $\forall a \in A, [\![a]\!]_I \neq c$. This approach would generate $O(|S|)$ test cases for each constant within a formula and is therefore too expensive to be feasible.
4. *L*ogical *C*onnector *R*eplacement (LCR): For predicates p and q, we have the two cases $p \wedge q \overset{\text{LCR}}{\rightarrowtail} p \vee q$ and $p \vee q \overset{\text{LCR}}{\rightarrowtail} p \wedge q$. In both cases we need test cases such that $[\![p]\!]_I = \neg[\![q]\!]_I$.
5. *R*e*F*erence *R*eplacement (RFR): For this operator we have $a_1 \overset{\text{RFR}}{\rightarrowtail} a_2$ where a_1 and

a_2 are references. In order for a test case to be able to kill such a mutant, we need the additional constraint $[\![a_1]\!]_I \neq [\![a_2]\!]_I$. Along similar lines to the case with the CRR operator, without additional information about the mutation, we would need test cases for $\forall a_i \in A \wedge i \neq 2, [\![a_i]\!]_I \neq [\![a_2]\!]_I$. Once again, this approach would require too much testing effort from users since it would give rise to $O(|S|)$ test cases.

6. *R*elational *O*perator *R*eplacement (ROR): The ROR operator works on sub formulas f and g operated on by a relational operator from $\{<, >, \leq, \geq, =\}$. We show examples of possible mutations and the corresponding constraints that can be included in the test generation to kill the mutants.
   - $f < g \overset{\text{ROR}}{\rightarrowtail} f > g$ or $f > g \overset{\text{ROR}}{\rightarrowtail} f < g$ can be killed by including the constraint $[\![f]\!]_I \neq [\![g]\!]_I$.
   - $f < g \overset{\text{ROR}}{\rightarrowtail} f \leq g$, $f > g \overset{\text{ROR}}{\rightarrowtail} f \geq g$, $f < g \overset{\text{ROR}}{\rightarrowtail} f \geq g$, or $f > g \overset{\text{ROR}}{\rightarrowtail} f \leq g$ can all be killed by including the constraint $[\![f]\!]_I = [\![g]\!]_I$.

The additional constraints described above simply refine a test case that exercises a du pair such that it also kills one or more mutants generated by the operators. In cases where the additional constraints make the constraint sets for a du pair unsolvable, they have to be ignored. In other cases, the "refinement" of the test case by the additional constraints (targeted at a specific operator) might result in some other kinds of mutants not being killed. These trade-offs have to be analyzed further. Ideally, performing these replacements (preferably through automatic test-case generation) would improve the mutation adequacy of a du-adequate test suite without an increase in the number of tests, thereby keeping the execution time of the test suite constant.

8.3.6 Discussion

As discussed at the beginning of this chapter, the primary goal of mutation testing is to evaluate the effectiveness of test suites at killing mutants. In this section, we have described how the collection of mutation operators we developed have helped us evaluate the mutation adequacy of du adequate test suites generated by the automatic test-case generator AutoTest. Analysis of the surviving mutants have also suggested the possibility of including additional constraints, in some cases, to make the generated test suites more mutation adequate.

8.4 Evaluation of GoalDebug

GoalDebug (described in Chapter 7) is a spreadsheet debugger that allows the user to mark cells as incorrect and specify the expected outputs.[2] The system converts the user expectations into constraints and generates a list of change suggestions, which when applied, would result in the expected outputs in the cells. Since many suggestions are generated in general, the user would have to go through and pick the correct one (or the most appropriate one and modify it as needed). To minimize the effort required of the user, we have developed a set of ranking heuristics that rank the suggestions before presenting them to the user. As far as the performance of GoalDebug is concerned, an *effective change suggestion* is one which would correct the error, and an *effective set of ranking heuristics* is one which would assign a high rank (1 ideally) to the correct

[2]The first version of GoalDebug was used in the preliminary evaluation described in Section 8.4.2. The system was then extended based on the results of the evaluation. It is the extended system that is described in Chapter 7.

change suggestion consistently. We carried out the evaluation described in this section to identify and correct weaknesses in GoalDebug so that the system would be more effective against a wide range of faults.

8.4.1 Experiment Setup

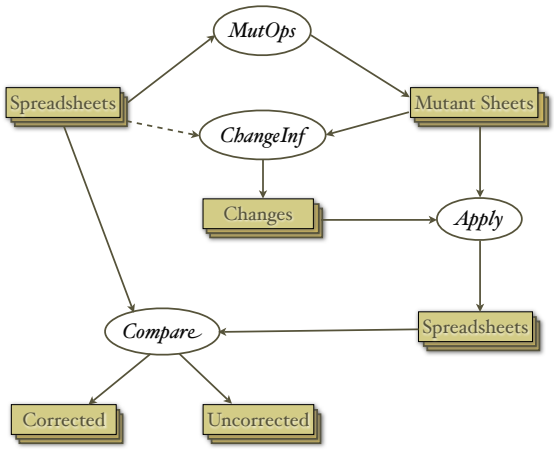


Figure 8.1: Setup for evaluating change suggestions

The experimental setup used to evaluate GoalDebug is shown in Figure 8.1. We start with a set of spreadsheets and use the collection of mutation operators to generate a (much larger) set of mutant spreadsheets. We use the output values in the cells in the original spreadsheets to specify the output expectations for the mutant cells. The change inference process is carried out on the mutant spreadsheet to generate a list of change suggestions. In cases in which any one of the change suggestions reversed the effect of the mutation, we also recorded the rank of the suggestion.

The values in the input cells (D_S) remain the same since we only consider mutations of the formula cells (P_S). Consider a mutation $f \overset{o}{\rightarrowtail} f'$ that acts on the formula at address a within the original spreadsheet S to generate the mutated spreadsheet S' which has the formula f' at address a. In the context of this evaluation the mutation is said to be *reversible* if $[\![f]\!]_S \neq [\![f']\!]_{S'}$. That is, the mutation is reversible only if the output from the mutated sheet differs from that of the original sheet, and the output from the original sheet is specified as the expectation for the cell with the mutated formula. In cases where $[\![f]\!]_S = [\![f']\!]_{S'}$, the output of the mutated sheet is the same as that of the original sheet for the same set of inputs. *Irreversible* mutants are formulas that evaluate to the same value as the original formula. Thus they cannot produce failures that can be observed by the user for the given set of inputs since a different expectation cannot be specified for the mutated formula. GoalDebug is principally inapplicable in those cases and cannot be invoked to generate change suggestions since the computed output and expected output are the same.

To study the effectiveness of change inference and ranking mechanisms of GoalDebug, we used spreadsheets from the evaluation of AutoTest together with a few we have used in other studies. Table 8.10 shows the summary information about the spreadsheets used in the evaluation. For each of the spreadsheets, the following information is given in the table.

1. Number of formula cells in the spreadsheet (Fml).
2. Total number of cells in the spreadsheet (Total).
3. The number of irreversible mutants that were generated.
4. The number of reversible mutants that were generated.

5. Total number of mutants generated from the given sheet.

Sheet	Cells		Mutants		
	Fml	Total	Irreversible	Reversible	Total
Microgen	2	12	143	33	176
GradesNew	8	26	157	181	338
FitMachine	6	18	366	74	440
Digits	6	14	172	293	465
NetPay	6	18	61	47	108
Purchase	15	50	172	153	325
RandJury	21	58	578	308	886
Sales	16	29	0	338	338
Solution	3	12	119	116	235
Budget	6	24	46	112	158
MBTI	28	83	902	243	1145
NewClock	10	24	156	165	321
GradesBig	21	48	283	647	930
Harvest	9	26	10	221	231
Payroll	54	100	347	1057	1404
Total	211	542	3512	3988	7500

Table 8.10: Sheets used in the evaluation

In general, not all mutation operators are applicable to all formulas. Moreover, for the purpose of this study, we excluded FDL (formula-deletion operator) and FRC (formula replace with constant operator) from the collection of operators. The FDL operator was excluded since it seems unlikely that a user would mark an empty cell as incorrect and specify the expected output. Along similar lines, the FRC operator was excluded since the mutation is not a minor change to reverse. To be able to reverse a FRC or FDL mutation, given a cell and an expected output, GoalDebug would have to come up with a formula that would compute the expected output in the marked cell. In practice, GoalDebug could accommodate these mutations by searching for candidate formulas within the spreadsheet, which if copied and pasted to the cell, would result in the ex-

pected value. The formula changes could then be ranked on the basis of similarity of regions or distance from the target cell. However, these mutations were not included since they are impractical from a debugging point of view.

8.4.2 Preliminary Evaluation

The number of mutants that have not been corrected by the first version of GoalDebug are shown in Table 8.11. The results of the preliminary evaluation showed that GoalDebug was ineffective against a wide variety of spreadsheet faults. All of the uncorrected mutants are created by the nine operators shown in Table 8.11. The reason for this observation is that GoalDebug did not have any change inference mechanism to reverse the errors seeded by the following operators: AOR, CRR, LCR, NRE, NRS, and ROR. Therefore, to make GoalDebug more effective, we needed to extend the system to include these classes of errors. We also needed to improve the system's coverage on the errors seeded by the CRP, RFR, and RRR operators.

Based on the preliminary evaluation, we identified the following two areas in which GoalDebug could be improved.

1. The change-inference mechanism needed to be substantially expanded to include coverage for a wider range of error situations the system could recover from. Without this extension, GoalDebug would be very limited in its applicability to real-world error situations. The downside of extending the change-inference mechanism is that a much higher number of change suggestions would get generated.

Sheet	Operators: Uncorrected [Total]								
	AOR	CRP	CRR	LCR	NRE	NRS	RFR	ROR	RRR
Microgen	6 [6]	0 [3]	4 [4]	1 [1]	0 [0]	0 [0]	3 [16]	3 [3]	0 [0]
GradesNew	18 [18]	0 [4]	25 [25]	1 [1]	0 [0]	0 [0]	16 [123]	10 [10]	0 [0]
FitMachine	9 [9]	3 [6]	11[11]	1 [1]	0 [0]	0 [0]	5 [41]	6 [6]	0 [0]
Digits	62 [62]	0 [17]	43 [43]	0 [0]	0 [0]	0 [0]	27 [152]	19 [19]	0 [0]
NetPay	3 [3]	0 [6]	12 [12]	0 [0]	0 [0]	0 [0]	0 [22]	4 [4]	0 [0]
Purchase	14 [14]	0 [4]	29 [29]	0 [0]	0 [0]	0 [0]	3 [91]	15 [15]	0 [0]
RandJury	87 [87]	0 [47]	33 [33]	6 [6]	0 [0]	0 [0]	6 [119]	16 [16]	0 [0]
Sales	72 [72]	0 [12]	49 [49]	0 [0]	0 [0]	0 [0]	24 [205]	0 [0]	0 [0]
Solution	21 [21]	0 [2]	12 [12]	0 [0]	0 [0]	0 [0]	11 [77]	4 [4]	0 [0]
Budget	15 [15]	0 [1]	18 [18]	0 [0]	0 [0]	0 [0]	10 [75]	3 [3]	0 [0]
MBTI	43 [43]	0 [24]	35 [35]	16 [16]	0 [0]	0 [0]	3 [120]	5 [5]	0 [0]
NewClock	20 [20]	1 [11]	22 [22]	1 [1]	0 [0]	0 [0]	0 [97]	14 [14]	0 [0]
GradesBig	6 [6]	3 [5]	28 [28]	1 [1]	99 [99]	27 [27]	12 [103]	14 [14]	272 [364]
Harvest	0 [0]	0 [0]	5 [5]	0 [0]	40 [40]	18 [18]	0 [24]	0 [0]	99 [134]
Payroll	170 [170]	0 [42]	166 [166]	0 [0]	0 [0]	0 [0]	39 [641]	38 [38]	0 [0]
Total	546 [546]	7 [184]	492 [492]	27 [27]	139 [139]	45 [45]	159 [1906]	151 [151]	371 [498]

Table 8.11: Original version of GoalDebug's effectiveness at correcting mutations

2. To keep the user effort to a minimum even after extending the change-inference mechanism, we would have to carry out enhancements to the ranking heuristics so that the system performs well even with the higher number of generated change suggestions.

8.4.3 Evaluation of the Extended System

We repeated the experiment after extending GoalDebug.[3] The effectiveness scores for the new version of the system are shown in Table 8.12. For each of the nine mutation operators, the percentage coverage of the old system (in yellow/lighter shade) is shown against the coverage of the new system (in blue/darker shade) in Figure 8.2. The results show that the extensions to the change-inference mechanism have increased the ability

[3]As explained at the beginning of Section 8.4, the extended version of the change-inference mechanism, together with the modified ranking heuristics, are described in Chapter 7.

of GoalDebug to recover from a much wider spectrum of errors.

We ran the Wilcoxon test to compare the old ranking heuristics against the new ones. The test shows that the new heuristics perform significantly better than the old ones ($p < 0.001$). Ideally, the correct change suggestion should be ranked within the top five ranks. The difference in ranks assigned by the two techniques is more important at high ranks than low ones. For example, a difference of 5 between ranks 1 and 6 is more important than a difference of 5 between ranks 100 and 105. To take this effect into consideration. we also ran statistical tests on the reciprocals of the ranks generated by the two techniques. Once again, the Wilcoxon test showed that the new set of ranking heuristics significantly outperforms the old one ($p < 0.001$). We also compared the performance of the two sets of ranking heuristics on mutants generated by each of the mutation operators. As can be seen from the p-values reported in the table on the left, the new heuristics significantly outperform the old ones for all operators.

Operator	p
AOR	< 0.001
CRP	< 0.001
CRR	< 0.001
LCR	0.008
NRE	0.036
NRS	0.005
RFR	< 0.001
ROR	< 0.001
RRR	< 0.001

The cumulative coverage percentages across ranks for the new heuristics (in dark red) are shown against those for the old (in light blue) in Figure 8.3. With the new heuristics in effect, the top ranked suggestion corrects the mutations in 59% of the cases, the top two suggestions correct the mutations in 71% of the cases, and so on. Putting the numbers in perspective, out of 3988 reversible mutants that were generated, the suggestion that corrects a given mutation is ranked in the top five suggestions in 80% of the cases with the new ranking heuristics as opposed to only 67% of the cases with the

Sheet	Operators: Uncorrected [Total]								
	AOR	CRP	CRR	LCR	NRE	NRS	RFR	ROR	RRR
Microgen	0 [6]	0 [3]	0 [4]	0 [1]	0 [0]	0 [0]	0 [16]	0 [3]	0 [0]
GradesNew	0 [18]	0 [4]	0 [25]	0 [1]	0 [0]	0 [0]	0 [123]	0 [10]	0 [0]
FitMachine	0 [9]	2 [6]	0 [11]	0 [1]	0 [0]	0 [0]	0 [41]	0 [6]	0 [0]
Digits	0 [62]	0 [17]	0 [43]	0 [0]	0 [0]	0 [0]	0 [152]	0 [19]	0 [0]
NetPay	0 [3]	0 [6]	0 [12]	0 [0]	0 [0]	0 [0]	0 [22]	0 [4]	0 [0]
Purchase	0 [14]	0 [4]	0 [29]	0 [0]	0 [0]	0 [0]	0 [91]	0 [15]	0 [0]
RandJury	0 [87]	0 [47]	0 [33]	0 [6]	0 [0]	0 [0]	0 [119]	0 [16]	0 [0]
Sales	0 [72]	0 [12]	0 [49]	0 [0]	0 [0]	0 [0]	0 [205]	0 [0]	0 [0]
Solution	0 [21]	0 [2]	0 [12]	0 [0]	0 [0]	0 [0]	0 [77]	0 [4]	0 [0]
Budget	0 [15]	0 [1]	0 [18]	0 [0]	0 [0]	0 [0]	0 [75]	0 [3]	0 [0]
MBTI	0 [43]	0 [24]	0 [35]	0 [16]	0 [0]	0 [0]	0 [120]	0 [5]	0 [0]
NewClock	0 [20]	1 [11]	0 [22]	0 [1]	0 [0]	0 [0]	0 [97]	0 [14]	0 [0]
GradesBig	0 [6]	1 [5]	0 [28]	0 [1]	23 [99]	0 [27]	0 [103]	5 [14]	73 [364]
Harvest	0 [0]	0 [0]	0 [5]	0 [0]	0 [40]	0 [18]	0 [24]	0 [0]	10 [134]
Payroll	0 [170]	0 [42]	0 [166]	0 [0]	0 [0]	0 [0]	0 [641]	0 [38]	0 [0]
Total	0 [546]	4 [184]	0 [492]	0 [27]	23 [139]	0 [45]	0 [1906]	5 [151]	83 [498]

Table 8.12: GoalDebug's effectiveness at correcting mutations after enhancements

old version of the ranking heuristics.

8.4.4 Discussion

The evaluation of GoalDebug using the collection of mutation operators helped us identify classes of errors GoalDebug was not effective against. We were able to use this information to extend the change-inference mechanism of GoalDebug. Furthermore, it also enabled us to refine the ranking heuristics. Since the collection of mutation operators has been designed to reflect real-world faults, the evaluation and subsequent modifications of the system have made GoalDebug effective against a wider range of potential errors.

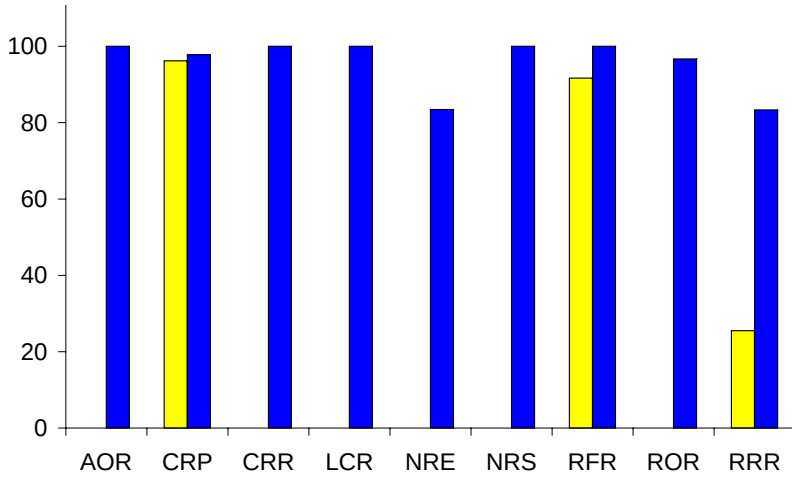


Figure 8.2: Comparison of coverage%

8.5 Evaluation of Combined Reasoning Approaches

None of the systems described in Chapter 3 protects the user from all spreadsheet errors. Each system has its own strengths and weaknesses. There is also a cost-benefit tradeoff associated with their use. We used the collection of mutation operators to study the effects of combining UCheck (automatic consistency checker described in Chapter 5) and WYSIWYT (framework for testing spreadsheets described in Section 3.4) to find out whether a combination of the two systems could result in an overall improved approach to error detection. We used the mutation operators to seed errors within spreadsheets used in the study. User interactions with WYSIWYT were simulated using a model we built based on empirical data of user interactions collected from a previous study [103]. Using this approach, we were able to combine WYSIWYT's reasoning, based on the spreadsheet and the (simulated) user inputs, with UCheck's reasoning, which is based

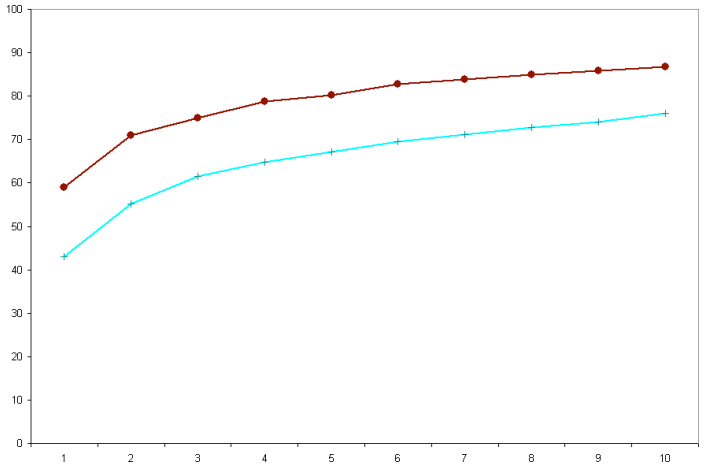


Figure 8.3: Cumulative coverage of ranking heuristics

solely on the data and formulas within the spreadsheet. We were therefore able to evaluate the error detection and reporting mechanisms of the two systems independently and together over a wide range of possible error situations.

Both WYSIWYT and UCheck use cell shading to convey fault-localization information to the user. WYSIWYT has a six-point scale of shading—unshaded (or zero, indicating no error) to darkest shade (or five, indicating high likelihood of error). UCheck uses dark shading for cells that are the sites of unit errors (local unit errors), light shading for cells that have reference to cells that have unit errors (propagation unit errors), and it leaves unshaded cells that have no unit errors. Prior empirical results have shown that end users consistently debug cells that have the darkest shading [103]. Therefore it is very important to ensure that the cells shaded the darkest are actually the sites of errors. As described in [103], there are two aspects to any error reporting mechanism. First, an *information base* consists of the information used to locate faults, and second,

a *mapping* determines how the information base is actually transformed into fault localization feedback and presented to the user. Both aspects of fault localization need to be integrated effectively for combining different reasoning approaches. That is, we need to come up with heuristics to combine the fault detection mechanisms of the two systems, and we also need to design heuristics to map this information into visual feedback.

To integrate UCheck's three-point scale with scores on WYSIWYT's six-point scale, we assigned a score of zero to unshaded cells, one to cells that are shaded yellow (since it is likely that the propagation errors might be masking local unit errors in these cells), and five to cells shaded orange by UCheck since the system is reporting unit errors with 100% confidence in these cases. The degree of shading of a cell c is returned by the function $score(c)$. Intuitively, the fault-localization feedback provided to the user can be considered to be more effective when the faulty cells are all shaded dark, and the non-faulty cells are shaded (if at all) light. We compute the *visual effectiveness (VE)* of the feedback in terms of the set of formula cells with correct and incorrect formulas, and the degree of shading [94].

$$VE = \sum_{c \in Faulty} \frac{score(c)}{|Faulty|} - \sum_{c \in Correct} \frac{score(c)}{|Correct|}$$

In the evaluation, we compared three heuristics for combining the feedback computed independently by UCheck and WYSIWYT.

1. *ComboMax* returns the darkest shading received from either of the two systems for each of the cells.
2. *ComboAverage* returns the average of the scores of the shading computed by the

two systems.

3. *ComboMin* returns the lightest cell shading received from either of the two systems.

We also considered two different mappings from the information base. For the *original mapping*, we used the scores from the actual shading generated by the two systems. For the *threshold mapping* we ignored cells shaded yellow by UCheck and cells with very low fault likelihood as reported by WYSIWYT. The rest of the cells were treated as if they were shaded with the darkest hue. In both mappings, we considered the three different strategies (*ComboMax*, *ComboAverage*, and *ComboMin*) for combining feedback.

8.5.1 Experiment Setup

The architecture of the combined reasoning system is shown in Figure 8.4. Excel acts as the user interface and captures the cell edits and user judgments about the correctness of the cells and sends the information to the reasoning database. UCheck uses the information about the spreadsheet to carry out unit checking and updates the reasoning database with the unit error information. WYSIWYT uses the information about the spreadsheet and the user actions and updates the reasoning database with the fault likelihoods of the cells. The fault-localization feedback that is reported to the user is based on the inputs from both UCheck and WYSIWYT.

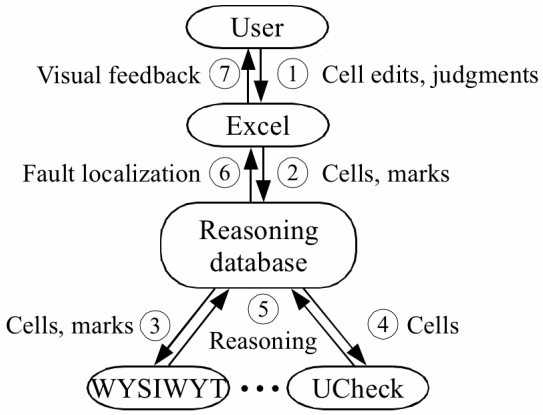


Figure 8.4: Combining reasoning from UCheck and WYSIWYT

8.5.2 Results

The results of our evaluation show that UCheck and WYSIWYT differ in their ability to detect various types of faults. UCheck is very effective at detecting reference mutations, whereas WYSIWYT detects a broader range of faults. Therefore, UCheck and WYSIWYT complement each other in their fault-detection capabilities.

The visual effectiveness scores for the various combination heuristics we considered are shown, sorted in descending order, in tables 8.13 (original mapping) and 8.14 (threshold mapping). We see that *ComboMax* scores the highest visual effectiveness scores for both mappings.

8.5.3 Discussion

UCheck and WYSIWYT support different approaches to spreadsheet development. UCheck's effectiveness is dependent on the labels entered by the user and their hierar-

Reasoning	Faulty	–	Correct	=	VE
Combo Max	1.730	–	0.119	=	1.611
UCheck	1.526	–	0.042	=	1.484
Combo Average	1.063	–	0.064	=	0.999
WYSIWYT	0.600	–	0.085	=	0.515
Combo Min	0.396	–	0.008	=	0.388

Table 8.13: Visual Effectiveness scores with original mapping

Reasoning	Faulty	–	Correct	=	VE
Combo Max	4.043	–	0.166	=	3.877
UCheck	3.814	–	0.106	=	3.708
Combo Average	2.339	–	0.093	=	2.246
WYSIWYT	0.864	–	0.079	=	0.785
Combo Min	0.636	–	0.019	=	0.617

Table 8.14: Visual Effectiveness scores with threshold mapping

chical structure in relationship to values and formulas. WYSIWYT, on the other hand, relies on the data-flow relationships between the cells and the user judgments about the correctness of cell outputs.

If the user is willing to invest time upfront in labeling the data, UCheck offers consistency checking for free, which leaves fewer number of errors the user would have to debug later. However, as can be seen from the comparison shown in Table 8.15, UCheck is only effective against errors seeded by a small subset of the mutation operators whereas WYSIWYT is effective against all of them.

If the user does not use labels within the sheet (or puts off labeling for later), WYSIWYT can help identify faults through testing. However, in this case, it is likely that a much greater effort would be required to identify all faults through testing alone. As far as design of fault localization feedback is concerned, *ComboMin* would be the strategy to adopt to provide conservative feedback, whereas, *ComboMax* would be the

Contributor	CRP	AOR	CRR	RFR	NRE	RRR	NRS	ROR	LCR
Neither	0	2	11	35	0	7	14	1	3
WYSIWYT only	3	7	13	50	0	2	13	5	0
Both				25	38	125			
UCheck only				26	85	235			

Table 8.15: Observed failures vs. reasoning

ideal approach if higher visual effectiveness is the goal.

In the evaluation of combining the reasoning from UCheck with that from WYSIWYT, the mutation operators allowed us to generate spreadsheets with a wide range of faults in them. Since the evaluation was done using a model of user interactions based on empirical data from previous studies, we were also able to simulate user behavior assuming varying levels of expertise. Such an extensive analysis would be prohibitively expensive if we were limited to actual user studies.

8.6 Conclusions and Future Work

In this chapter we have presented a collection of mutation operators for spreadsheets. The operators are designed to mimic end-user programming errors, and have been used in the following applications.

1. The operators have been used to evaluate the effectiveness of du-adequate test suites generated by automatic test case generation system described in Chapter 6. The results of the evaluation suggested extensions to the constraint-generation mechanism of AutoTest that would make the generated test suites du and mutation adequate.

2. The collection of mutation operators has been used to seed errors in spreadsheets to carry out evaluations of other spreadsheet systems we have developed. These evaluations have led to refinements of the systems studied and also show the utility of the mutation operators in carrying out empirical evaluations.

The list of spreadsheet mutation operators proposed in Section 8.2 is by no means complete. We suspect the list will evolve as we try to mirror more errors from real-world spreadsheets. A more complete list of mutation operators would allow a practitioner to adapt the operators to other applications more easily.

Chapter 9 – Conclusions

> *Standing barefoot in a river of clues, most people would not get their toes wet.*
>
> Brian Kantor

Spreadsheets are very easy to use for simple tasks. However, development and maintenance tasks become rather difficult and error prone with increasing complexity of the spreadsheet. In this thesis, we have described methods aimed at helping end users develop safe spreadsheets. Each one of the approaches has an associated cost-benefit tradeoff. The costs in this case are the demands on the users' effort, time, and expertise. The benefits are the guarantees a system (or a combination of systems) can provide in terms of the level of correctness of spreadsheets.

Systems like XeLda [18] and Slate [35] require the user to annotate cells with unit information. These systems then carry out unit checking (based on physical units and those derived from labels) and alert the user to the presence of unit errors in their spreadsheets. UCheck [1] carries out consistency checking only on the basis of units derived from headers. However, UCheck offers the advantage that it is fully automatic.

An end user carrying out ad hoc testing of a spreadsheet gets few benefits in return for carrying out a high-cost activity. Testing within a framework like WYSIWYT [100], on the other hand, is more focused and the users get feedback on their progress. Moreover, the use of a test-case generator like HMT [50] or AutoTest [4] lowers the effort

required of the user. Integrating the testing framework with a spreadsheet debugger like GoalDebug can further lower the cost encountered by users in locating and correcting faults within their spreadsheet programs.

Spreadsheet formulas should not just be viewed in isolation. The spatial arrangement of the cells within the spreadsheet implicitly captures a lot of information about the user's mental model. We have been able to exploit the spatial structure directly in automatic header inference [1] and in automatic template inference [5], and indirectly in designing the change-inference mechanism and ranking heuristics of GoalDebug [2].

Studies have shown that spreadsheets typically contain cells that are copies of each other in the sense that the formula in one was created by a copy-paste action of the formula in the other, or by copy-paste followed by slight modifications [52, 80]. Such related cells can be grouped together into regions, and the similarity in formulas can be exploited in inferring templates [5], minimizing testing effort [31], and goal-directed debugging [2]. Another insight gleaned from the study of regions within spreadsheets is that many tasks performed during the construction of spreadsheets are repetitive. These tasks include insertion or deletion of rows or columns, copying of singleton cells or entire blocks of cells, and modifications to formulas and input cells. This insight led to the design of the ViTSL/Gencel framework [12, 43, 44] which supports the creation and maintenance of spreadsheets free from reference, type, and range errors through the use of spreadsheet templates.

All the approaches we have worked on thus far carry out their inferences from static analyses of a given spreadsheet. We do not exploit information about the steps involved in the creation of any given spreadsheet. This information could be very useful in in-

ferring the user's mental model. Moreover, we also have chosen not to solicit extra information from the user explicitly. The main reason for these choices was because we wanted to explore the design space of approaches based on static analyses as much as possible before we require additional input from the user.

As we have demonstrated, static analyses can go a long way in assisting end users in creating correct spreadsheets. Future work on integrating more interactive features promises more effective approaches. Our work with spreadsheets has shown that the benefits of sound software engineering concepts can indeed be brought within the grasp of end users to help them develop effective software.

Bibliography

[1] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets Through Spatial Analyses. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.

[2] R. Abraham and M. Erwig. Goal-Directed Debugging of Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 37–44, 2005.

[3] R. Abraham and M. Erwig. How to Communicate Unit Error Messages in Spreadsheets. In *1st Workshop on End-User Software Engineering*, pages 52–56, 2005.

[4] R. Abraham and M. Erwig. AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 43–50, 2006.

[5] R. Abraham and M. Erwig. Inferring Templates from Spreadsheets. In *28th IEEE Int. Conf. on Software Engineering*, pages 182–191, 2006.

[6] R. Abraham and M. Erwig. Type Inference for Spreadsheets. In *ACM Int. Symp. on Principles and Practice of Declarative Programming*, pages 73–84, 2006.

[7] R. Abraham and M. Erwig. A Specifications-Based Approach to Correct Spreadsheets. 2007. Submitted for publication.

[8] R. Abraham and M. Erwig. A Type System Based on End-User Vocabulary. 2007. Submitted for publication.

[9] R. Abraham and M. Erwig. GoalDebug: A Spreadsheet Debugger for End Users. In *29th Int. Conf. on Software Engineering*, 2007. to appear.

[10] R. Abraham and M. Erwig. Mutation Operators for Spreadsheets. 2007. Submitted for publication.

[11] R. Abraham and M. Erwig. UCheck: A Spreadsheet Unit Checker for End Users. *Journal of Visual Languages and Computing*, 18(1):71–95, 2007.

[12] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual Specifications of Correct Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 189–196, 2005.

[13] Y. Adachi. Intellisheet: A Spreadsheet System Expanded by Including Constraint Solvers. *IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, page 173, 2001.

[14] AgentSheets. `http://www.agentsheets.com/`.

[15] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A Type System for Statically Detecting Spreadsheet Errors. In *18th IEEE Int. Conf. on Automated Software Engineering*, pages 174–183, 2003.

[16] E. E. Allen, D. Chase, V. Luchangco, J. Maessen, and G. L. Steele. Object-oriented Units of Measurement. In *Object-oriented Programming Languages, Systems, and Applications*, pages 384–403, 2004.

[17] C. Allwood. Error Detection Processes in Statistical Problem Solving. *Cognitive Science*, 8(4):413–437, 1984.

[18] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the Unit Correctness of Spreadsheet Programs. In *26th IEEE Int. Conf. on Software Engineering*, pages 439–448, 2004.

[19] Y. Ayalew, M. Clermont, and R. Mittermeir. Detecting Errors in Spreadsheets. *Spreadsheet Risks, Audit and Development Methods*, 1:51–62, 2000.

[20] D. Ballinger, R. Biddle, and J. Noble. Spreadsheet Visualization to Improve End-User Understanding. In *Australian Symposium on Information Visualization*, pages 99–109, 2003.

[21] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[22] A. Blackwell. First Steps in Programming: A Rationale for Attention Investment Models. In *IEEE Symp. on Human-Centric Computing Languages and Environments*, pages 2–10, 2002.

[23] A. F. Blackwell and T. R. G. Green. Notational Systems - The Cognitive Dimensions of Notations Framework. *HCI Models, Theories, and Frameworks: Toward and Interdisciplinary Science*, pages 103–133, 2003.

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

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

[26] T. A. Budd and D. Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18(1):31–45, 1982.

[27] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. In *ACM Symp. on Principles of Programming Languages*, pages 220–233, 1980.

[28] M. M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming*, 11:155–206, March 2001.

[29] M. M. Burnett, C. Cook, J. Summet, G. Rothermel, and C. Wallace. End-User Software Engineering with Assertions. In *25th IEEE Int. Conf. on Software Engineering*, pages 93–103, 2003.

[30] M. M. Burnett and M. Erwig. Visually Customizing Inference Rules About Apples and Oranges. In *2nd IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pages 140–148, 2002.

[31] M. M. Burnett, A. Sheretov, B. Ren, and G. Rothermel. Testing Homogeneous Spreadsheet Grids with the "What You See Is What You Test" Methodology. *IEEE Transactions on Software Engineering*, 29(6):576–594, 2002.

[32] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. on Software Engineering*, 2(3):215–222, 1976.

[33] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A Comparison of Data Flow Path Selection Criteria. In *8th Int. Conf. on Software Engineering*, pages 244–251, 1985.

[34] M. Clermont, C. Hanin, and R. Mittermeir. A Spreadsheet Auditing Tool Evaluated in an Industrial Context. *Spreadsheet Risks, Audit and Development Methods*, 3:35–46, 2002.

[35] M. J. Coblenz, A. J. Ko, and B. A. Myers. Using Objects of Measurement to Detect Spreadsheet Errors. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 314–316, 2005.

[36] Stagecast Creator. `http://www.acypher.com/creator/`.

[37] D. Cullen. Excel snafu costs firm $24M. *The Register*, June 2003.

[38] J. S. Davis. Tools For Spreadsheet Auditing. *Int. J. of Human-Computer Studies*, 45(4):429–442, 1996.

[39] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help For the Practicing Programmer. *IEEE Computer*, 11(4):34–41, 1978.

[40] Dreamweaver. `http://www.adobe.com/products/dreamweaver/`.

[41] A. Ebrahimi. Novice Programmer Errors: Language Constructs and Plan Composition. *Int. Journal of Human-Computer Studies*, 41(4):457–480, 1994.

[42] G. Engels and M. Erwig. ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications. In *20th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 124–133, 2005.

[43] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic Generation and Maintenance of Correct Spreadsheets. In *27th IEEE Int. Conf. on Software Engineering*, pages 136–145, 2005.

[44] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein. Gencel — A Program Generator for Correct Spreadsheets. *Journal of Functional Programming*, 16(3):293–325, 2006.

[45] M. Erwig and M. M. Burnett. Adding Apples and Oranges. In *4th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pages 173–191, 2002.

[46] EUSES. End Users Shaping Effective Software. `http://EUSESconsortium.org`.

[47] EuSpRIG. European Spreadsheet Risks Interest Group. `http://www.eusprig.org/`.

[48] R. Ferguson and B. Korel. The Chaining Approach For Software Test Data Generation. *ACM Trans. on Software Engineering and Methodology*, 5(1):63–86, 1996.

[49] G. Filby. *Spreadsheets in Science and Engineering*. Springer, 1995.

[50] M. Fisher, G. Rothermel, D. Brown, M. Cao, C. Cook, and B. Burnett. Integrating Automated Test Generation into the WYSIWYT Spreadsheet Testing Methodology. *ACM Trans. on Software Engineering and Methodology*, 15:150–194, 2006.

[51] M. Fisher II, M. Cao, G. Rothermel, C. Cook, and M. M. Burnett. Automated Test Case Generation for Spreadsheets. In *24th IEEE Int. Conf. on Software Engineering*, pages 141–151, 2002.

[52] M. Fisher II and G. Rothermel. The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanism. In *1st Workshop on End-User Software Engineering*, pages 47–51, 2005.

[53] Fortress Programming Language. `http://research.sun.com/projects/plrg/`.

[54] Microsoft Frontpage. `http://www.microsoft.com/frontpage/`.

[55] J. D. Gannon. An Experimental Evaluation of Data Type Conventions. *Communications of the ACM*, 20(8):584–595, 1977.

[56] A. Givens. Brookhaven Corruption Probe, Investigators Made Mistakes, March 2004. `Newsday.com`.

[57] Gnome. Gnome Gnumeric. `http://www.gnome.org/projects/gnumeric/`.

[58] K. Godfrey. Computing Error at Fidelity's Magellan Fund. In *Forum on Risks to the Public in Computers and Related Systems*, January 1995.

[59] J. B. Goodenough and S. L. Gerhart. Toward a Theory of Test Data Selection. In *Proc. of the Int. Conf. on Reliable Software*, pages 493–510, 1975.

[60] Google. Google Spreadsheets. `http://docs.google.com/`.

[61] R. G. Hamlet. Testing Programs With the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[62] R. L. Hayen and R. M. Peters. How to Ensure Spreadsheet Integrity. *Management Accounting*, 60(9):30–33, 1989.

[63] K. S. How Tai Wah. A Theoretical Study of Fault Coupling. *Software Testing, Verification and Reliability*, 10(1):3–45, 2000.

[64] K. S. How Tai Wah. An Analysis of the Coupling Effect I: Single Test Data. *Science of Computer Programming*, 48(2-3):119–161, 2003.

[65] W. E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Trans. on Software Engineering*, 2(3):208–215, 1976.

[66] T. Igarashi, J. Mackinlay, B.-W. Chang, and P. T. Zelleweger. Fluid Visualization of Spreadsheet Structures. In *IEEE Symp. on Visual Languages*, 1998.

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

[68] L. Jiang and Z. Su. Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs. In *28th Int. Conf. on Software Engineering*, pages 262–271, 2006.

[69] A. Kennedy. Dimension Types. In *5th European Symp. on Programming*, LNCS 788, pages 348–362, 1994.

[70] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *27th Int. Conf. on Software Engineering*, pages 126–135, 2005.

[71] A. J. Ko and B. A. Myers. Development and Evaluation of a Modem of Programming Errors. In *IEEE Symp. on Human-Centric Computing Languages and Environments*, pages 7–14, 2003.

[72] A. J. Ko and B. A. Myers. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Int. Conf. on Human Factors in Computing Systems*, pages 151–158, 2004.

[73] J. Lawrence, R. Abraham, M. M. Burnett, and M. Erwig. Sharing Reasoning about Faults in Spreadsheets: An Empirical Study. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 35–42, 2006.

[74] J. Lawrence, R. Abraham, M. M. Burnett, and M. Erwig. Toward Sharing Reasoning to Improve Fault Localization in Spreadsheets. In *2nd Workshop on End-User Software Engineering*, 2006.

[75] J. F. Lerch, M. M. Mantei, and J. R. Olson. Skilled Financial Planning: The Cost of Translating Ideas Into Action. *ACM Conf. on Human Factors in Computing Systems*, pages 121–126, 1989.

[76] C. Lewis and G. M. Olson. Can Principles of Cognition Lower the Barriers to Programming? In *2nd Workshop on Empirical Studies of Programmers*, pages 248–263, 1987.

[77] B. Lisper and J. Malmstrom. Haxcel: A Spreadsheet Interface to Haskell. In *Proc. Int. Workshop on the Implementation of Functional Languages*, pages 206–222, September 2002.

[78] D. Mason and D. Keane. Spreadsheet Modeling in Practice: Solution or Problem. In *Interface*, pages 82–84, 1989.

[79] Microsoft. Microsoft Excel. `http://office.microsoft.com/excel/`.

[80] R. Mittermeir and M. Clermont. Finding High-Level Structures in Spreadsheet Programs. In *9th Working Conference on Reverse Engineering*, pages 221–232, 2002.

[81] G. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[82] A. J. Offutt. Investigations Of The Software Testing Coupling Effect. *ACM Trans. on Software Engineering and Methodology*, 1(1):5–20, 1992.

[83] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination Of Sufficient Mutant Operators. *ACM Trans. on Software Engineering and Methodology*, 5(2):99–118, 1996.

[84] 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, 1996.

[85] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the Orthogonal. In *Proc. Mutation*, pages 45–55, 2000.

[86] OpenOffice. OpenOffice Calc. `http://www.openoffice.org/product/calc.html`.

[87] J. Paine. ModelMaster: A Tool For Spreadsheet Intelligibility, Spreadsheet Integrity, and Spreadsheets on the Web. `http://www.j-paine.org/mm_publications.html/`.

[88] J. F. Pane, C. A. Ratanamahatana, and B. A. Myers. Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *Int. Journal of Human-Computer Studies*, 54:237–264, 2001.

[89] 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.

[90] R. R. Panko. Applying Code Inspection to Spreadsheet Testing. *Journal of Management Information Systems*, 16(2):159–176, 1999.

[91] R. R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. In *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.

[92] R. R. Panko and R. P. Halverson, Jr. Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks. In *29th Hawaii Int. Conf. on System Sciences*, 1996.

[93] S. L. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003.

[94] A. Phalgune, C. Kissinger, M. Burnett, C. Cook, L. Beckwith, and J. Ruthruff. Garbage In, Garbage Out? An Empirical Look at Oracle Mistakes by End-User Programmers. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 45–52, 2005.

[95] S. Prabhakarao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and Behaviors of End-User Programmers with Interactive Fault Localization. In *IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pages 203–210, 2003.

[96] L. Prechelt and W. F. Tichy. A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking. *IEEE Transactions on Software Engineering*, 24(4):302–312, 1998.

[97] K. Rajalingham, D. Chadwick, B. Knight, and D. Edwards. Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development. In *33rd Hawaii Int. Conf. on System Sciences*, pages 1–9, 2000.

[98] K. Rajalingham, D. R. Chadwick, and B. Knight. Classification of Spreadsheet Errors. *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2001.

[99] M. Rosson and J. M. Carroll. The Reuse of Uses in Smalltalk Programming. *ACM Trans. on Computer-Human Interaction*, 3(3):219–253, 1996.

[100] G. Rothermel, M. M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A Methodology for Testing Spreadsheets. *ACM Transactions on Software Engineering and Methodology*, pages 110–147, 2001.

[101] J. Ruthruff, E. Creswick, M. M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main. End-User Software Visualizations for Fault Localization. In *ACM Symp. on Software Visualization*, pages 123–132, 2003.

[102] J. R. Ruthruff and M. M. Burnett. Six Challenges in Supporting End-User Debugging. In *1st Workshop on End-User Software Engineering*, pages 57–61, 2005.

[103] J. R. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick, and M. M. Burnett. Interactive, Visual Fault Localization Support for End-User Programmers. *Journal of Visual Languages and Computing*, 16:3–40, 2005.

[104] P. Saariluoma and J. Sajaniemi. Transforming Verbal Descriptions Into Mathematical Formulas in Spreadsheet Calculation. *Int. Journal of Human-Computer Studies*, 41:915–948, 1994.

[105] J. Sajaniemi. Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *Journal of Visual Languages and Computing*, 11:49–82, 2000.

[106] C. Scaffidi, M. Shaw, and B. Myers. Estimating the Numbers of End Users and End User Programmers. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.

[107] A. Scott. Shurgard Stock Dives After Auditor Quits Over Company's Accounting. *The Seattle Times*, November 2003.

[108] P. Sestoft. CoreCalc: A Spreadsheet Core Implementation in C#. `http://www.itu.dk/people/sestoft/corecalc/`.

[109] M. Shaw. Avoiding Costly Errors in Your Spreadsheets. *Contractor's Management Report*, 11:2–4, 2004.

[110] M. G. Simkin. How to Validate Spreadsheets. *Journal of Accountancy*, 180(11):130–138, 1987.

[111] J. C. Spohrer and E. Soloway. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM*, 29(7):624–632, 1986.

[112] J. G. Spohrer and E. Soloway. Analyzing the High Frequency Bugs in Novice Programs. In *First Workshop on Empirical Studies of Programmers*, pages 230–251, 1986.

[113] M. Stadelmann. A Spreadsheet Based on Constraints. In *ACM Symp. on User Interface Software and Technology*, pages 217–224, 1993.

[114] G. Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology*, RTI Project Number 7007.011, 2002.

[115] T. Teo and M. Tan. Quantitative and Qualitative Errors in Spreadsheet Development. In *30th Hawaii Int. Conf. on System Sciences*, pages 25–38, 1997.

[116] T. Teo and M. Tan. Spreadsheet Development and 'What-if' Analysis: Quantitative Versus Qualitative Errors. *Accounting Management and Information Technologies*, 9:141–160, 2000.

[117] M. Tukiainen. Uncovering Effects of Programming Paradigms: Errors in Two Spreadsheet Systems. In *12th Workshop of the Psychology of Programming Interest Group (PPIG)*, pages 247–266, 2000.

[118] U.S. Department of Education. Audit of the Colorado Student Loan Program's Establishment and Use of Federal and Operating Funds for the Federal Family Education Loan Program, July 2003. Report ED-OIG/A07-C0009.

[119] U.S. Department of Health and Human Services. Review of Medicare Bad Debts at Pitt County Memorial Hospital, January 2003. Report A-04-02-02016.

[120] E. J. Weyuker. More Experience with Data Flow Testing. *IEEE Trans. on Software Engineering*, 19(9):912–919, 1993.

[121] E. J. Weyuker, S. N. Weiss, and D. Hamlet. Comparison of Program Testing Strategies. In *Fourth Symposium on Software Testing, Analysis and Verification*, pages 154–164, 1991.

[122] D. Whittaker. Spreadsheet Errors And Techniques For Finding Them. *Management Accounting*, 77(9):50–51, 1999.

[123] T. Williams. Spreadsheet Standards. *Tech. Report*, 1987.

[124] B. T. Vander Zanden, D. Baker, and J. Jin. An Explanation-Based, Visual Debugger for One-Way Constraints. In *17th Annual ACM Symp. on User Interface Software and Technology*, pages 207–216, 2004.