2017

Spreadsheet Tools for Data Analysts

Daniel W. Barowy
University of Massachusetts Amherst

SPREADSHEET TOOLS FOR DATA ANALYSTS

A Dissertation Presented

by

DANIEL W. BAROWY

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2017

College of Information and Computer Sciences

SPREADSHEET TOOLS FOR DATA ANALYSTS

A Dissertation Presented

by

DANIEL W. BAROWY

Approved as to style and content by:

Emery D. Berger, Chair

Michael Ash, Member

Andrew McGregor, Member

Alexandra Meliou, Member

James Allan, Chair
College of Information and Computer Sciences

ACKNOWLEDGMENT

First, I must thank my parents, Bill and Judy. You instilled me with the desire to be a scientist at an early age[1]. Second, thanks to my little brother, Adam. You enrolled in a graduate program *before* me, engendering a great deal of envy on my part. Clearly you were having more fun. Thanks to you, I realized that a major life change was in order.

Many people helped me through the winding path to graduate school. Thanks to Peter D'Errico and Alan Gaitenby of the UMass Legal Studies Department, and Jae Young Lee, Wayne Snyder, and John Day of the Boston University Metropolitan College Computer Science Department.

I am honored to consider many fellow graduate students my friends: Charlie Curtsinger, John Altidor, Kaituo Li, John Vilk, Tongping Liu, Marianna Rapoport, Emma Tosch, Rian Shambaugh, Bobby Powers, and Sam Baxter. I hope that all of you appreciate the extent to which you made coming to work every day fun.

To Leeanne Leclerc and the UMass CS administrative staff: you rock! In addition to ensuring that I always dotted my i's and crossed my t's, you endured my many requests for equipment (e.g., projectors), frequently for questionable extracurricular activities (e.g., "PLASMA Movie Night").

Many thanks to my mentors and collaborators Ben Zorn, Sumit Gulwani, Ted Hart, Eric Chung, Daniel Goldstein, Siddharth Suri, and Rodric Rabbah from whom I learned how to be a professional computer scientist. Thanks to the faculty who liberally offered me their advice and encouragement over the years, especially David

[1]Anybody needing proof may examine my marked up copy of Dr. Seuss' *My Book About Me*.

Jensen, Andrew McGregor, Yuriy Brun, Alexandra Meliou, Scott Kaplan, Steve Freund, Margaret Robinson, Heather Pon-Barry, and Jack Wileden.

I am incredibly fortunate to have ended up with Emery Berger as my advisor. Emery is much more than a research mentor. He pushed me to improve myself in ways I was neither aware of nor fully appreciated until afterward. Emery, working with you was fun and hard—exactly what I wanted from graduate school. Thank you.

Finally, my wife Karen deserves more thanks than anybody. Karen, you and I dreamed up the "Dan and Karen plan" a long time ago, and we stuck with it. I think that this is more of a testament to your character than it is to mine. For example, you are serenely dealing with the fact *as I write this* that I neglected (again) to make dinner. Newton once said "If I have seen further than others, it is by standing upon the shoulders of giants." Having never married, Newton had a poor selection of giants. You are my giant. I honestly do not know how I could have done this without you.

ABSTRACT

SPREADSHEET TOOLS FOR DATA ANALYSTS

SEPTEMBER 2017

DANIEL W. BAROWY

B.Sc., BOSTON UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Emery D. Berger

Spreadsheets are a natural fit for data analysis, combining a simple data storage and presentation layer with a programming language and basic debugging tools. Because spreadsheets are accessible and flexible, they are used by both novices and experts. Consequently, spreadsheets are hugely popular, with more than 750 million copies of Microsoft Excel installed worldwide. This popularity means that spreadsheets are the most popular programming language on the planet and the *de facto* tool for data analysis.

Nevertheless, spreadsheets do not address a number of important tasks in a typical analyst's pipeline, and their design frequently complicates them. This thesis describes three key challenges for analysts using spreadsheets. 1) *Data wrangling* is the process of converting or mapping data from a "raw" form into another form suitable for use with automated tools. 2) *Data cleaning* is the process of locating and correcting omitted or erroneous data. 3) *Formula auditing* is the process of finding and correcting

spreadsheet program errors. These three tasks combined are estimated to occupy more than three quarters of a data analyst's time. Furthermore, errors not caught during these steps have led to catastrophically bad decisions resulting in billions of dollars in losses. Advances in automated techniques for these tasks may result in dramatic savings in both time and money.

Three novel programming language-based techniques were created to address these key tasks. The first, *automatic layout transformation using examples*, is a program synthesis-based technique that lets spreadsheet users perform data wrangling tasks automatically, at scale, and without programming. The second, *data debugging*, is technique for data cleaning that combines program analysis and statistical analysis to automatically find likely data errors. The third, *spatio-structural program analysis* unifies positional and dependence information and finds spreadsheet errors using a kind of anomaly analysis.

Each technique was implemented as an end-user tool—FlashRelate, CheckCell, and ExceLint respectively—in the form of a point-and-click plugin for Microsoft Excel. Our evaluation demonstrates that these techniques substantially improve user efficiency. Finally, because these tools build on each other in a complementary fashion, data analysts can run data wrangling, cleaning, and formula auditing tasks together in a single analysis pipeline.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

INTRODUCTION

Spreadsheets are one of the most popular applications on the planet, with more than 750 million estimated users of Microsoft Excel alone. Their point-and-click interface and intuitive interaction model means that novice users can start using them quickly and with little training. Their apparently simplicity belies a sophisticated dynamic, functional programming language, a reactive presentation layer, automated processing and debugging facilities, and many other features. Given these features, spreadsheets are also appealing to many experts, and are therefore the *de facto* tool of choice for data analysis.

Unfortunately, a number of key tasks in a typical data analysis pipeline are not well supported by spreadsheets. This dissertation addresses three tasks that remain challenging in the context of spreadsheets: *data wrangling*, *data cleaning*, and *formula auditing*. Collectively, these three tasks comprise the majority of a data analysts's time.

0.0.1 Data wrangling

The first problem arises because spreadsheets allow users great flexibility in storing data. This flexibility comes at a price: users often treat spreadsheets as a poor man's database, leading to creative solutions for storing high-dimensional data. The trouble occurs when users need to answer queries. Data manipulation tools make strong assumptions about data layouts and cannot read these ad hoc databases. Converting data into an appropriate layout, a task called *data wrangling*, requires programming skills or a major investment in manual reformatting.

0.0.2 Data cleaning

The second problem is that data inputs frequently contain errors and omissions. Addressing these data issues, a task called *data cleaning*, is necessary to ensure that the ensuing calculations are correct. Finding data errors is complicated by the size of datasets and the fact that it is typically impossible to know a priori which inputs are wrong, since data rarely comes with precise specifications. Worse, some calculations are sensitive to input errors while others are not, meaning that there is not necessarily a correlation between the magnitude of the input error and the effect that error has on the calculation that depends on it.

0.0.3 Formula auditing

The third problem is that, like ordinary programs, spreadsheet programs frequently contain programming errors. Spreadsheet environments encourage poor programming practices. For example, spreadsheets lack high-level abstractions like user-defined functions, encouraging programmers to manually copy formulas to perform repeated calculations, so the number of formulas scales with the input data. Consequently, it is increasingly difficult to ensure correctness as spreadsheets grow.

0.0.4 Contributions

This dissertation presents three techniques that address the key data analysis tasks outlined above. Collectively, these techniques substantially reduce the effort required for data analysts to produce large-scale, bug-free spreadsheets.

1. **Automatic layout transformation using examples** is a program synthesis-based technique that lets ordinary users convert ad hoc layouts into relational tables without programming. Instead, users supply examples of the desired output rows. At its core is FLARE, a novel extraction language that extends regular expressions with geometric constructs. A novel program synthesis

algorithm generates FLARE programs in seconds from a small set of examples in a wide variety of real-world scenarios. FLASHRELATE, an interactive tool for Microsoft Excel, builds on these core techniques, letting end users extract data by pointing and clicking.

2. **Data debugging** is an approach that combines program analysis and statistical analysis to automatically find *likely data errors*. Testing and static analysis can help root out bugs in programs, but not in data. Since it is impossible to know a priori whether data are erroneous, data debugging instead locates data that has a disproportionate impact on the computation. Such data is either very important or wrong. CHECKCELL, the first data debugging tool for Microsoft Excel, was built to evaluate this approach. CHECKCELL is more precise and efficient than standard outlier detection techniques, and it automatically identifies a key flaw in the infamous Reinhart and Rogoff spreadsheet.

3. **Spatio-structural program analysis** is a hybrid statistical and static program analysis for spreadsheet formulas, designed to locate *data reference errors*, a common form of spreadsheet program bug. The analysis leverages the observation from conventional programming languages that anomalous code is often wrong. Spatio-structural analysis extends this finding to spreadsheet programs by learning likely data reference invariants; deviations from these invariants often indicates a data reference error. We built EXCELINT, a spatio-structural bug finding tool for Microsoft Excel. Evaluated against 29 real-world spreadsheets, EXCELINT dramatically reduces the effort needed to audit spreadsheets and it improves on the state of the art by providing a high precision, user-friendly tool that finds three times the number of reference bugs identified by previous state of the art tools.

CHAPTER 1

BACKGROUND

Despite the availability of tools and techniques for programming data analyses using spreadsheets, this task remains a challenge. An end-to-end view of the data analysis pipeline is illustrative. Problems often start at the very beginning, when importing data. Users must choose a layout for their data, often before they know exactly how the data will be used. Revising these layouts, in order to make an analysis possible, is called *data wrangling*. Next, data quality must be addressed as users discover that their data contains errors, missing values, and other irregularities. This activity is called *data cleaning*. After writing analysis code, programs are often flawed. Programmers remove these errors by *debugging* their programs. Programmers rarely perform these activities in a linear fashion, revisiting the same steps over and over.

Data wrangling and data cleaning are estimated to comprise anywhere from 50-80% of an analyst's time [65]. Debugging is conservatively estimated to comprise 50% of an analyst's remaining time [95]. *At best*, an analyst spends a meager 25% of their time actually developing the desired data analysis code. Even worse, errors sometimes go completely unnoticed during development, meaning that flawed analyses are produced and relied upon for critical decision-making.

This dissertation aims to improve analyst productivity and effectiveness by introducing techniques designed to reduce the amount of time spent doing these ancillary activities and to support the development of correct analyses. The following sections expand on these problems and detail prior attempts at resolving them.

	A	B	C	D	E
1		value	year	value	year
2	Albania	1000	1950	930	1981
3	Austria	3139	1951	3177	1955
4	Belgium	541	1947	601	1950
5	Bulgaria	2964	1947	1959	1958
6	Czech ...	2416	1950	2503	1960

...

Figure 1.1: A semi-structured spreadsheet excerpt with two sample tuples highlighted. The first tuple (red) represents the timber harvest (per 1000 hectares) for Albania in 1950. The second tuple (blue) represents the timber harvest for Austria in 1950.

1.1 Data wrangling: Automation is layout sensitive

Spreadsheets combine their data model and view, a combination that gives spreadsheet creators a high degree of freedom when laying out their data. Although spreadsheets are tabular and ostensibly *two-dimensional tables*, end users may store any high-dimensional data in a spreadsheet as long as they can devise a spatial layout (using, e.g., headers, whitespace, and relative positioning) that projects that data into two dimensions. As a result, while spreadsheets allow compact and intuitive visual representations of data well suited for human understanding, their flexibility complicates the use of powerful data-manipulation tools (e.g., relational queries) that expect data in a certain form. We refer to these spreadsheets as *semi-structured* because their data is in a regular format that is nonetheless inaccessible to data-processing tools. Unless semi-structured data can be decoded into the appropriate *normal form* expected by these tools, data is effectively "locked-in" to the user's format.

This lock-in problem is widespread. Research suggests that few spreadsheets are trivially convertible to database relations. Only 22% of 200 randomly-chosen spreadsheets scraped from the web can be converted by performing "Export as CSV." [23] For a user with data trapped in one of these formats, little hope is available in the form of off-the-shelf tools.

Country	Harvest	Date
Albania	1000	1950
Albania	930	1981
...		
Austria	3139	1951
Austria	3177	1955
...		
Belgium	541	1947
Belgium	601	1950
...		

Figure 1.2: A normalized spreadsheet excerpt with two sample tuples highlighted. The highlighted tuples are the same as shown in Figure 1.1.

The spreadsheet shown in Figure 1.1 pairs timber harvest values with their year in a structure that repeats to the right for each country listed in the column on the left. While this spreadsheet may be convenient to answer certain queries[1], for others it presents difficulties. For example, the simple query "How many years did the timber harvest exceed 2000?" cannot be answered without complex selection logic like the kind shown in the Visual Basic for Applications program in Figure 1.3. By contrast, the same data, laid out in the form shown in Figure 1.2, trivially answers the same query using Microsoft Excel's point-and-click "wizard" tools.

Users are faced with a difficult choice. One option is to choose the layout best suited for processing *before* any data entry has started. The other option is to reformat data after entry. Assuming that the user even knows what queries they intend to run, the first option makes processing easy but may require data entry using an inconvenient form. Furthermore, knowing the right database "normal form" requires formal training in database theory, which is not likely to be common given that the majority of spreadsheet users are not professional programmers. The alternative facilitates data entry and may be visually appealing, but users must either write

[1]Aside from storing data, this spreadsheet's purpose is not clear to this author.

```
Function Extract() As Collection
  ...
  Set Tuples = New Collection
  rYear.Pattern = "^19[0-9]{2}$"
  rValue.Pattern = "^[0-9]+$"
  For Each ws In Worksheets
    For Each cell In ws.UsedRange
      x = cell.Column
      y = cell.Row
      x_rt = x + 1
      If rYear.Test(cell.Value) _
        And rValue.Test(_
          ws.Cells(y, x_rt).Value) Then
        Dim tupleCoords
        tupleCoords = Array(ws.Index, x, _
                            y, x_rt, y)
        Tuples.Add (cellCoords)
      End If
    Next
  Next
  Extract = Tuples
End Function
```

Figure 1.3: A Visual Basic for Applications program for extracting spreadsheet data into a normal form. This program contains navigation logic and must maintain a tabular data structure (a collection of tuples). This program produces the extraction shown in Figure 1.2. Variable declarations and driver code are omitted for brevity.

selection code—well beyond the capabilities of most spreadsheet users—or be willing to manually reformat (i.e., re-enter) the data. Given that many users make the wrong layout choice with respect to automation, many spreadsheets are effectively "locked" into *ad hoc* formats.

Finally, it is frequently the case that analysts are forced to work with data in the form that is given to them. For instance, the U.S. federal government's Open Data Policy transparency initiative requires that all newly-generated data to be machine readable, and except in certain circumstances, publicly available. As of the time of this dissertation, there are more than 15,000 datasets in the CSV spreadsheet format available on their website, `data.gov`. Nonetheless, given the set of queries that a user has in mind, this freely-accessible data may not be simple to query without first changing the layout. Although `data.gov` is a large-scale effort to make data available, this data is still inaccessible to a large number of interested parties—e.g., journalists, policy makers, and educators—because these parties may not possess skills needed to automatically reformat the data.

1.1.1 Related Work

There are many approaches to data wrangling. By contrast to the techniques described below, this dissertation proposes a solution for *arbitrary* transformations (relying on no database of patterns) that need not be hierarchical. We believe that this dissertation presents strong evidence that the appropriate transformation schema is task-dependent; in other words, there is no single "best" transformation. Finally, our approach does not require that users understand the transformation logic itself, which may be complex. Users need only understand the end result of transformation logic, instead driving the transformation process by supplying sample *transformed outputs*.

1.1.1.1 Programming-By-Example

The area of programming by example (which includes FlashRelate) promises to enhance productivity for end users [46, 64]. The most closely related work is ProgFromEx [50], which performs tabular transforms for spreadsheets already in tabular format using a version-space algebra and input-output example pairs. Another recent technology, Quicksilver [66], synthesizes relational algebra queries over normalized spreadsheet tables. Quicksilver cannot handle any of the transformation tasks in our benchmarks. FlashExtract is a by-example framework for extracting data from text files [63]. FlashExtract uses only relative string positions, so it cannot make use of 2D spatial information (including the motivating example in Fig. 1.1). FlashExtract programs are strictly hierarchical, so multiple constraints cannot refer to "overlapping" regions.

1.1.1.2 Language Approaches

SXPath [75] includes spatial primitives in its queries. The PADS project simplifies ad hoc data-processing tasks for programmers by developing DSLs and learning algorithms to extract data from textual formats [40]. OpenRefine helps users clean and transform their spreadsheet data into relational form, but requires that users program [89]. None of these synthesize extraction programs.

1.1.1.3 Algorithmic Approaches

An important related body of work focuses on extracting relational data from data on web pages [16,39]. While SILA [74] defines spatial abstractions like FlashRelate, it extracts records algorithmically, and not from examples. Gyro [52] expresses spatial constraints in the form of geometric regions. Gyro's inference algorithm is based on searching a database of known programs.

1.1.1.4 Machine Learning

Wrappers are procedures to extract data from Internet resources. *Wrapper induction* automatically constructs such wrappers [62]. There has been a wide variety of work in this area, ranging from supervised systems [57, 62, 72], semi-supervised systems [19], to unsupervised systems [27]. SENBAZURU [21] automatically infers *hierarchical structure* in spreadsheets using a set of classifiers. By contrast, FLASHRELATE can be used to perform *arbitrary* extraction tasks from *arbitrary* spreadsheets. HAEXCEL [28] focuses on recovering the true relational schema from the spreadsheet data. WRANGLER automatically infers likely transformation rules and presents them in natural language. Two pitfalls with WRANGLER are that users must be understand the *potential* effects of the available transforms, and must be capable of finding alternate transforms in the event that the inference is wrong.

1.2 Data cleaning: Input errors

After an analyst has the data in the right form, it must be inspected to ensure that is error-free. Regardless of the source of the data, data errors are common. Prior work has shown that data errors often arise from the following sources [51]: 1) Data entry errors, including typographical errors and transcription errors from illegible text. 2) Measurement errors, when the data source itself, such as a disk or a sensor, is faulty or corrupted. 3) Data integration errors, where inconsistencies arise due to the mixing of different data, including unit of measurement mismatches.

For typographical errors alone, users make a mistake for every 0.5% to 1.23% of characters typed on average [11, 77]. For spreadsheets, this translates to data cells containing errors at a rate of approximately 5% [77, 78]. Unfortunately, this means that for spreadsheets of even a modest size (at least 50 cells), at least one error is highly likely (72%) [76, 78].

Manual checking scales poorly with data input size. Existing automatic approaches to data cleaning include *statistical outlier detection*, *cross validation*, and *input validation*. Outlier detection reports data anomalies by comparing them to a known distribution. Spreadsheets do not supply information about input distributions and it is unlikely the spreadsheet users know them. Cross-validation requires fault-free comparison data, which may difficult or impossible to procure. Programmers perform input validation by writing validation routines that mechanically check that inputs match a specification. Beside being difficult to define precisely—data rarely comes with a precise specification—no commonly-used spreadsheet package currently has this capability.

Finally, none of these techniques capture an entire class of important subtle errors: inputs that would pass automated checking but that nonetheless cause unusual program behavior. Depending on the computation, an input error could be an outlier that has no effect (e.g., `MIN()` of a set of inputs containing an erroneously large value), or a non-outlier that affects a computation dramatically (e.g., `IF A1 = 0, "All is Well", "Fire Missiles"`). Like regular programs, spreadsheets are often a mix of functions that consume and produce both numbers and strings. Automatic error-checking therefore must be capable of handling a wide variety of data types while ideally being sensitive to the effect the error has on the program.

1.2.1 Related Work

1.2.1.1 Database Cleaning

Most past work on locating or removing errors in data has focused on *data cleaning* for database systems [48, 80]. Standard approaches include statistical outlier analysis for removing noisy data [94], interpolation (e.g., with averages), and cross-correlation with other data sources [54].

A number of approaches have been developed that allow data cleaning to be expressed programmatically or applied interactively. Programmatic approaches include AJAX, which expresses a data cleaning program as a DAG of transformations from input to output [44]. Data Auditor applies rules and target relations entered by a programmer [45]. A similar domain-specific approach has been employed for data streams to smooth data temporally and isolate it spatially [59]. Potter's Wheel is an interactive tool that lets users visualize and apply data cleansing transformations [81].

To identify errors, Luebbers et al. describe an interactive data mining approach based on machine learning that builds decision trees from databases. It derives logical rules (e.g., "BRV = 404 $\Rightarrow$ GBM = 901") that hold for most of the database, and marks deviations as errors to be examined by a data quality engineer [67]. Raz et al. describe an approach aimed at arbitrary software that uses Daikon to infer invariants about numerical input data and then report discrepancies as "semantic anomalies" [35, 82]. Data debugging is orthogonal to these approaches: rather than searching for latent relationships in or across data, it measures the interaction of data with the programs that operate on them.

1.2.1.2 Statistical Outlier Analysis

Outlier analysis dates to the earliest days of statistics, with some of its earliest uses found in making nautical measurements more robust. Widely-used approaches include Chauvenet's criterion, Peirce's criterion, and Grubb's test for outliers [10]. For spreadsheets specifically, Benford's Law [73] is frequently used by forensic analysts to detect fraudulent data [15, 73]. All of these techniques are *parametric*: they require that the data belong to a known distribution, e.g., Gaussian (normal). Unfortunately, input data does not neatly fit into a predefined statistical distribution. Moreover, identifying outliers leads to false positives when they do not materially contribute to

the result of a computation (i.e., have no impact). By contrast, data debugging only reports data items with a substantial impact on a computation.

1.2.1.3 Sensitivity Analysis

Sensitivity analysis is a method used to determine how varying an input affects a model's range of outputs. Most sensitivity analyses are analytic techniques; however, the one-factor-at-a-time technique, which systematically explores the effect of a single parameter on a system of equations, is similar to data debugging in that it seeks to numerically approximate the effect of an input on an output. Recent research employing techniques from sensitivity analysis in static program analyses seeks to determine whether programs contain "discontinuities" that may indicate a lack of program robustness [4, 20, 47].

Data debugging differs from sensitivity analysis in two important respects. First, data debugging is a fully-automated black-box technique that requires only dependence information. Second, unlike sensitivity analysis, data debugging does not vary a parameter through a known range of valid values, which must be parameterized by an analyst. Instead, data debugging infers an output distribution via a nonparametric statistical approach that does not require analysts to supply parameters.

1.3 Formula auditing: Formula reference errors

The last step in an analyst's pipeline is to write code that performs an analysis. Like other programming tasks, data analyses can have bugs. Nonetheless, despite many years of research into automatically detecting and correcting programming errors, spreadsheet users still make programming logic errors at an alarmingly high rate. Recent work shows that 5% of all spreadsheet formulas contain an error, a figure that is consistent with traditional (non-spreadsheet) programs [77]. Many of these errors go uncaught because automated analysis tools for spreadsheets are particularly

ineffective. A recent study of commercial static analysis tools for spreadsheets found that they discovered formula errors at an extremely low rate: only 0.52% of errors were correctly identified [8].

An important class of formula errors are *reference errors*, an error that occurs when a formula fails to refer to the correct set of inputs. One way that reference errors occur is because of a shortcoming in spreadsheet languages: spreadsheets have no facility for function abstraction or subroutines. Users repeat calculations by copying and pasting formula strings. When a user fails to update references during this copy-and-paste workflow, reference errors occur. The likelihood of making an error increases as the size of repeated calculations grows, making these errors hard to find.

Excel's "formula fill" feature is intended to mitigate this problem by automatically copying and updating formulas. Unfortunately, to operate correctly, users must *manually* insert address mode annotations into the original formula. For example, when using formula fill to copy the formula `=A1 * B1` from cell `C1` to cells `C2, ..., C10` such that the copies refer to `B2, ..., B10` respectively, but that the reference to cell `A1` remains constant[2], users must first modify the original formula. Using Excel's "absolute addressing mode" annotation, `$`, the original formula should be changed to `=$A$1 * B1` before invoking formula fill. If this modification is not performed, formula fill *inserts* reference errors. For example, using the unmodified formula, it generates the formulas `=A2 * B2`, `=A3 * B3`, `=A4 * B4`, and so on.

Another form of reference error is a failure to refer to all of the relevant data in a range of cells. These kinds of omissions, referred to by one researcher as "the most dangerous type of error," are especially insidious because there is nothing obviously wrong with the spreadsheet [77]. Inadvertent omissions were a key contributor to the

[2] `A1` might literally be a constant, like π.

flawed conclusion in the infamous Reinhart-Rogoff spreadsheet, whose logic was used to justify flawed fiscal austerity measures in the European Union [55].

1.3.1 Spreadsheet Error Detection

The unconstrained nature of spreadsheets, in contrast to most conventional programming languages, complicates automatic error checking. Spreadsheets have only rudimentary dynamic datatypes, no facilities for structured or object-oriented programming, and no user-defined functions. Beyond obvious errors like unparseable formulas or math errors like division by zero, in general, knowing whether a given formula is in error requires knowing user intent, which is well beyond the capabilities of state of the art program analyses. While it is sometimes possible to glean this information from semantic cues, such as headers (e.g., a column with the header "Totals" probably contains numeric sums), this evidence is at best circumstantial.

Nonetheless, upon inspection, humans can often reason about user intent and find unambiguously incorrect programs that produce wrong output. This work refers to these bugs as *manifest errors*. A second, more pernicious form of error is a *latent error*, which produces the correct output, but only by accident. This second class of errors surfaces as manifest errors only later in the life cycle of a spreadsheet when data values are updated. *Reference errors*, the focus of this work, can be found both as manifest and latent errors.

Cell `H58` in Figure 1.4 shows a manifest error. The programmer's intent was to compute a row total, however an off-by-one reference error means that the total displayed is for the wrong row. Note that in Excel, this problem can only be spotted by first switching to "formula view" and then by clicking on the reference inside the formula, which highlights it. For large spreadsheets, this approach does not scale.

Cell `H57` in Figure 1.4 shows a latent error. While it happens to be true that the correct row total is zero, the value in `H7` is a hand-entered value, not computed. Were

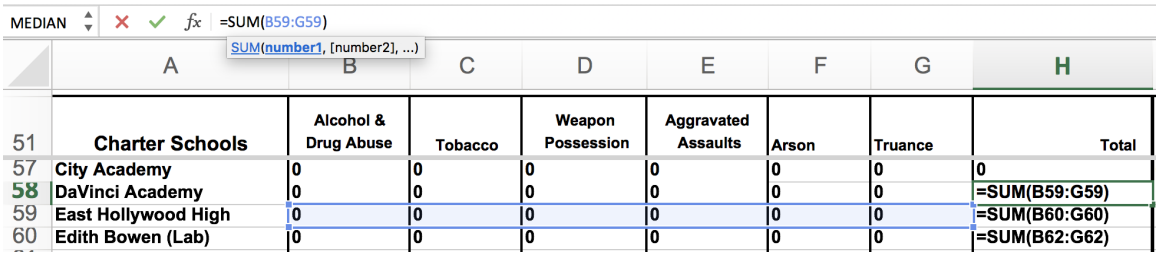


Figure 1.4: An example spreadsheet containing several errors. This spreadsheet was drawn from the FUSE corpus [9], shown here using Excel's "formula view". Cell `H58` is a *manifest error*, computing a total for the wrong row (`H59` and `H60` have the same bug). Cell `H57` shows a more subtle *latent error*—a missing formula—which will only appear when data in `B57` through `G57` are updated.

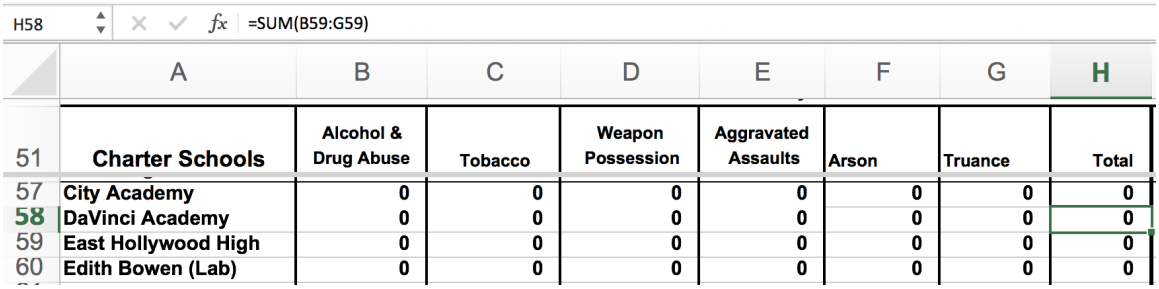


Figure 1.5: Errors are obscured in Excel's default view. The same spreadsheet shown in Figure 1.4. Errors are harder to spot because formulas are hidden.

the user of the spreadsheet to later update the values in `B57` through `G57`, the total would be wrong.

Neither of these errors are obvious as is shown in Figure 1.5, which shows that Excel's default view obscures them. In both cases, Excel's built-in error checking feature fails to flag these cells.

1.3.2 Related Work

Conventional error-checking tools for spreadsheets, like those found in Microsoft Excel, rely on *ad hoc* patterns that are believed to be indicative of errors. Recent work in the software engineering community focuses on detecting *spreadsheet smells*, which like error rules are anti-patterns that reflect violations of best practices. Much like source code "linters," flagged items are not necessarily errors. While this work discusses

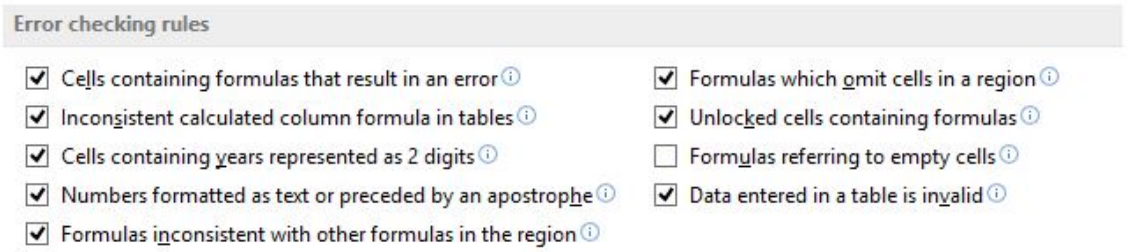


Figure 1.6: Error checking rules provided by Microsoft Excel 2013. See Chapter 1.3.2.2.

only one smell tool at length—CUSTODES—other tools adopt similar approaches [24, 30, 53]. We also discuss data type-based approaches and some approaches that exploit user markup for context. Finally, we discuss the anomaly-based approach that forms the basis for this dissertation.

1.3.2.1 Microsoft Excel

Figure 1.6 shows the error detection rules that Microsoft Excel employs [92], each of which may be selectively enabled or disabled. Excel's approach is based on recognizing common pre-defined patterns of errors and flagging specific cells that the patterns identify as potentially wrong.

The Excel user interface applies these rules and then uses an unobtrusive visual cue—a green triangle in the upper left corner of the cell—to highlight cells with potential issues[3]. The user can then consult a pop-up message that provides more details about the error along with a contextual menu that offers hardcoded fixes to the potential problem (e.g., to copy an adjacent formula into the cell). In addition, Excel provides a rudimentary formula dependence visualization, an "arrow" overlay pointing to (or from) a formula's (non-transitive) dependencies.

[3]Oddly, the slightly more intrusive red triangle sometimes seen in the upper right corner of a cell is reserved for user notes.

	A	B	C	D	E	F	G	H
51	Charter Schools	Alcohol & Drug Abuse	Tobacco	Weapon Possession	Aggravated Assaults	Arson	Truance	Total
52	AMES	0	0	0	0	0	0	0
53	American Leadership	0	0	0	0	0	0	0
54	American Prep Acad	0	0	0	0	0	0	0
55	Beehive Sci & Tech	0	0	0	0	0	0	0
56	Channing Hall	0	0	0	0	0	0	0

Figure 1.7: An Excel false positive. Excel flags the error in cell `H52` because it does not reference all available data in row 53 (see green triangle in upper left of cell). While `H52` is incorrect for other reasons (it references the wrong row altogether), this rule is a clear false positive.

These rules fall into several categories. Some rules check the contents of cells. For example, they check that the result of a computation is not itself an error (i.e., the expression cannot be evaluated), or that a value that appears to be a date is not represented as a 2-digit number. Other rules involve the relationship between cells. For example, formulas may omit cells in a contiguous region. The fixed set of rules that Excel employs is intended to cover real-world cases that are perceived by Excel's developers to be both common and likely to be actual errors.

Figure 1.7 shows an example of Excel's error reporting. In this example, Excel flags cell `H52` as problematic with a green triangle. The reason, according to Excel's error detector, is because the formula "omits adjacent cells." While the flagged cell is indeed problematic (as with the example in Figure 1.4, it sums the wrong row), Excel's fix is wrong: it suggests that the formula be updated to sum over `B53:N53`. Strangely, the fix itself violates Excel's rule that flags formulas that are inconsistent with other formulas in a region, since the "fixed" formula becomes the only such formula on the spreadsheet.

1.3.2.2 Spreadsheet "Smells"

CUSTODES[4] [24] is a command-line tool written in Java that identifies anti-patterns, commonly referred to as "code smells," using a combination of rules and cluster analysis. CUSTODES is available for free online and includes a data set of ground truth spreadsheets containing annotations for spreadsheet smells. CUSTODES attempts to detect four types of spreadsheet smells:

- **Missing Formula Smell.** Cells that contain hard-coded constants when neighboring cells in rows and columns contain formulas are suspicious [24, 30].
- **Dissimilar Reference Smell.** Cells that reference different cells than their neighbors are suspicious. The canonical example is an omitted cell from a range reference.
- **Dissimilar Operation Smell.** Cells that use different functions than their neighbors are suspicious. The canonical example is using `SUM` instead of `+`.
- **Hard-coded Constant Smell.** This is a more granular version of the *Missing Formula Smell*. A cell is marked as suspicious when its abstract syntax tree (AST) contains a hard-coded constant where a neighboring cell uses a function call.

CUSTODES is command-line tool written in Java. CUSTODES first clusters cells in a worksheet using features such as cell layout, formatting, and AST similarity. The tool then categorizes each cell, conditioned on its cluster. Note that CUSTODES outputs no smell labels: it only prints the set of "smelly" cells. CUSTODES also has no visualization. Users must consult the command-line output and compare it against their spreadsheet manually.

[4] http://sccpu2.cse.ust.hk/custodes/

Pattern-based and smell checkers are inherently limited. By definition, they cannot uncover errors that do not fall into an existing pattern, even if a spreadsheet contains unusual constructs. Cluster-based smell detectors like CUSTODES rely on rigid structures and strong consistency within logical spreadsheet regions. Unfortunately, Excel's rules can fail to detect many serious formula errors in real spreadsheets. Fundamentally, smell detectors like CUSTODES employ essentially the same approach as Excel's, but use some statistical approaches to "relax" the rules in order to improve recall.

Spreadsheet smells are largely orthogonal to spreadsheet anomaly detection: smells are not necessarily anomalous, and vice versa. Unlike smell detection approaches, our approach can detect arbitrary inconsistencies in spatial and structural reference patterns. Further, unlike CUSTODES, our approach is based entirely on a unified spatial and structural abstraction and does not rely on heuristics or domain knowledge.

1.3.2.3 Type and Unit Checking

Past work on detecting errors in spreadsheets has focused on inferring units and relationships (*has-a*, *is-a*) from information like structural clues and column headers, and then checking for inconsistencies [1, 3, 5, 18, 36, 37, 38, 60]. For example, XeLda checks that formulas process values with incorrect units or if derived units clash with unit annotations. There also has been considerable work on testing tools for spreadsheets [17, 41, 56, 60, 85, 86].

These analyses can and do find real bugs in spreadsheets, but they are largely orthogonal to the our approach In fact, many of the bugs that we find in this work would be considered type- and unit-safe.

1.3.2.4 Other Approaches to Spreadsheet Correctness

Other approaches to spreadsheet correctness include creating stronger specifications for a computation (such as creating models, using templates [2,33], or using declarative

approaches to describing the computation, etc.). While these approaches may be effective, they require additional programmer expertise, extra effort, and depending on the technique, may not be applicable to existing spreadsheets. This dissertation aims to support data analysis tasks for non-programmers and as many existing spreadsheets as possible to create the greatest benefit for the spreadsheet user community.

Another promising approach is to create more informative visualizations of the computation being performed so that the user can find errors by inspecting the visualization (e.g. [58] is one example). This dissertation explores several visualizations of reference behavior and of likely bugs. Many other visualizations are possible[5].

1.3.2.5 Anomaly Analysis

An alternative to pattern based bug detectors are anomaly-based approaches. Anomaly analysis leverages the observation from conventional programming languages that anomalous code is often wrong [25, 29, 34, 49, 82, 93]. This lets an analysis circumvent the difficulty of obtaining program correctness rules. For spreadsheets, rules used by pattern detectors are either very general (e.g., expressions must evaluate) or imprecise (e.g., adjacent cells should have the same formula). Anomaly analysis circumvents this problem by instead reasoning about the relative frequency of features found in code.

The canonical example of anomaly analysis is whether an omitted statement to release a lock on a data structure is an error or not [34]. Knowing whether the programmer intended to omit such a call requires knowing what effect the programmer *wanted* to obtain, information that is unavailable to any program analysis. Nonetheless, if in 999 out of 1,000 instances, `lock` statements are followed by an `unlock` statement, it is reasonable to conclude that the omitted `unlock` is likely to be an error. Hard-

[5]We explored many visualizations during the development of this dissertation, but only present the best visualizations here. Some interesting but ultimately sub-par visualizations included error heatmaps, reference "spectra", and reference "edge detection" plots.

coding such a relationship as a rule is likely to suffer from precision problems in general, as program control flow can sometimes obscure even rigidly-followed invariants. But learning such a rule from a *given* codebase adjusts for a programmer's idiosyncratic code constructs.

Instead, anomaly analysis mines likely invariants using generic templates—a kind of statistical feature—that describe relationships between program constructs. By comparing the relative frequencies of violations of the extracted rule, likely errors can be flagged without needing to know programmer intent. For example, a feature might be of the form *statement* `a` *is always followed by statement* `b`. The analysis does not need to know how to instantiate variables `a` and `b` a priori. The analysis then applies the template across a code base, learning both `a` and `b`. For example `a` could be "a `lock` statement" and `b` might be "an `unlock` statement." If such a relationship is as common as its contradiction ("a `lock` statement is *not* always followed by an `unlock` statement."), then the relationship is unlikely to be meaningful. However, if the relationship is frequently maintained with few violations, those violations likely indicate a real error.

Anomaly analysis was developed primarily for conventional programming languages like C or Java, not spreadsheets. This dissertation extends the anomaly-based approach to spreadsheets, in particular, by observing spatial relationships between formulas and their inputs in spreadsheet layouts.

CHAPTER 2

FLASHRELATE: EXTRACTING RELATIONAL DATA FROM SEMI-STRUCTURED SPREADSHEETS USING EXAMPLES

The previous chapter outlined challenges in three stages of data analysis. This chapter presents an automated approach designed to facilitate spreadsheet layout preparation for data analysts of all skill levels.

This chapter introduces *automatic layout transformation using examples*[1], a program synthesis-based technique that lets ordinary users transform structured relational data from spreadsheets *without programming*. At its core is FLARE, a novel transformation language that extends regular expressions with geometric constructs. Users transform data by supplying examples of output relational tuples, and the algorithm synthesizes a program guaranteed to satisfy those examples.

We built FLASHRELATE to evaluate our technique. This chapter makes the following contributions: 1) the identification of a critical step in data analysis that benefits from automation; 2) the FLARE domain-specific language and runtime that makes transformations possible; 3) the FLASHRELATE algorithm that generates correct FLARE programs in seconds from a small set of examples; 4) an intuitive point-and-click interface for layout transformation that requires no programming expertise; and 5) finally, demonstration that FLASHRELATE is effective for a large number of real-world scenarios, many of which are drawn from Excel user help forums.

[1]This work appeared at PLDI 2015 [12].

2.1 Approach

The FlashRelate tool relies on two core technologies to make automatic transformation of input spreadsheets possible. The first is a formal language called Flare that defines all possible layout transformation operations. The second is a search algorithm that finds a valid Flare program given a set of example tuples given by the user. The combination of formal language and search, a technique called *counterexample-guided inductive synthesis* (CEGIS), is known to be complete for the class of finite programs[2]. Nonetheless, naïve pairings of formal languages and search algorithms are not likely to produce useful program synthesizers, as CEGIS's combinatorial nature means that search may run for an unacceptably long amount of time. FlashRelate defines a *sufficiently expressive* formal language with a *domain-specific* search technique and search pruning heuristics. These refinements mean that despite the inherent combinatorial nature of the approach, satisfactory programs can often be found quickly (typically under 2 seconds) and with few input examples (typically 4). As a result, is it possible to build an interactive user interface on top of these core technologies that is fast enough to be useful, providing novice users with a practical technology for data wrangling tasks.

2.2 The Flare Transformation Language

Flare is a domain specific language designed to support transformations. A Flare program takes a spreadsheet as input and returns a set of tuples (a relational table) as output. The Flare language defines the set of transformations possible using the FlashRelate system.

The design of Flare is inspired by scripting languages with regular expression capabilities. While regular expressions are a powerful language-based mechanism for

[2] "A finite program is one whose input is bounded and which terminates on all inputs after a bounded number of operations." [88]. All Flare programs are finite.

```
    <prog>  ::=  <cpair> | <cpair> "[" <pseq> "]"
   <cpair>  ::=  <spatialc> <cellc>
    <pseq>  ::=  <cpair> | <cpair> "," <pseq>
   <cellc>  ::=  <regx> <anchor> | <capture>
 <capture>  ::=  "<" <name> "," <regx> ">" <anchor>
  <anchor>  ::=  ":" <spatialc> <regx> | ϵ_a
    <regx>  ::=  "/" String "/"
    <name>  ::=  String
<spatialc>  ::=  <vert> <horiz>
    <vert>  ::=  <vdir> <quant>
   <horiz>  ::=  <hdir> <quant>
    <vdir>  ::=  "u" | "d" | ϵ_v
    <hdir>  ::=  "l" | "r" | ϵ_h
   <quant>  ::=  "*" | "*?" | "*#"
              |  "+" | "+?" | "+#"
              |  "{" ℕ "}" | ϵ_q
```

Figure 2.1: FLARE syntax. FLARE extends regular expressions with spatial constraints. `String` represents any string literal. $\mathbb{N}$ represents a positive integer literal. FLARE is not sensitive to whitespace, except within regular expression `String` literals.

```
<Harvest,¬w>:u+/value/
[ r<Date,¬w>, l*<Country,αw> ]
```

Figure 2.2: A FLARE program that converts the table shown in Fig. 1.1 to the table shown in Fig. 1.2. The subexpression "αw" is shorthand for the regular expression that matches alphabetic characters and whitespace while "$\neg w$" is shorthand for the regular expression that matches any character except whitespace.

capturing and transforming string data, they are not powerful enough to capture relational information encoded in spreadsheets without additional support code. This is because spreadsheets are more than strings; they are strings embedded in a two-dimensional matrix. FLARE augments regular expressions with *geometric* constraints needed to capture and transform spreadsheet data into relational tables without support code. The key insight is that geometric constraints like direction and distance concisely express the spatial relationships underpinning the desired table extraction.

In a nutshell, the execution of a FLARE program constructively enumerates all possible combinations of cells from the input spreadsheet that satisfy the set of constraints described by the program. As with regular expressions, only those constructs that satisfy the constraints are candidates for transformation. FLARE has two kinds of constraints: 1) a *cell constraint* defines the set of valid strings belonging to a **column of the output table**, and 2) a *spatial constraint* defines the set of permissible spatial relationships **between two columns of the output table**. These two constraints may be composed according to the syntax rules shown in Fig. 2.1.

2.2.1 Example

The FLARE program written in Fig. 2.2 transforms the spreadsheet shown in Fig. 1.1 into the "long" format spreadsheet shown in Fig. 1.2. In general, the right layout depends on the query, so unless a user can anticipate all possible future queries, no perfectly-suited layout can be chosen a priori. With FLASHRELATE, this choice is unnecessary; layouts can always be changed. By asking the user for example transfor-

mations, FlashRelate automatically produces Flare transformation programs like the one shown in Fig. 2.2. FlashRelate frees the user to enter their data in any layout that they find convenient.

2.2.2 Constraints

The following section discusses each constraint and its details informally. A formal definition of Flare semantics may be found in Chapter A.

2.2.2.1 Cell Constraints

In Flare, cell constraints (`<cellc>` in Fig. 2.1) are essentially regular expressions[3]. Regular expressions are enclosed by a pair of slashes, `//`. For example, the expression `/^[0-9]+$/` matches a string that contains only numbers. From here on, we refer to this expression as `Num`.

Informally, a cell constraint is a boolean function that takes a spreadsheet and a cell location as parameters and returns *true* if and only if the constraint is satisfied by the cell at that location. Lines 6 and 7 in Fig. A.2 in the Appendix describe the meaning of cell constraints formally.

As with many regular expression implementations, a Flare cell constraint may either merely constrain or *constrain and capture* data (`<capture>` in Fig. 2.1). Capturing means that cell data is returned in an output tuple in addition to being used as a constraint. Capture is denoted by enclosing a cell constraint within a pair of angle brackets, `<>`. A valid Flare program must contain at least one capturing cell constraint. A captured cell must also be associated with a `<name>`; this name defines an *output column*.

---

[3] FlashRelate uses C#'s PCRE-like regular expressions [70].

2.2.2.2 Spatial Constraints

Spatial constraints describe the relative spatial relationships between two output columns (`<spatialc>`, Fig. 2.1) Spatial constraints are also the mechanism used to compose two subprograms. An n-tuple subprogram composed with an m-tuple subprogram yields an $n + m$-tuple subprogram.

The essential component of a spatial constraint is a *geometric descriptor* (`<vdir>` and `<hdir>` in Fig. 2.1). There are four basic descriptors, *up*, *down*, *left*, and *right*, denoted by `"u"`, `"d"`, `"l"`, and `"r"` respectively. A spatial constraint may contain up to two geometric descriptors, one for the vertical direction, and one for the horizontal direction. One may informally think of a spatial constraint as a boolean function that takes two cell locations as parameters and returns *true* if and only if those two cells satisfy the specified geometry in the spreadsheet. Line 11 in Fig. A.2 in the Appendix describes the meaning of spatial constraints formally.

Spatial constraints are directed, declaring that a functional spatial relationship exists between a *parent* cell and *child* cell. For example, a child cell is "to the left" of the parent cell. Strictly speaking, the *child-of* operator declares a spatial relationship between the cells yielded by two FLARE subprograms (2^{nd} rule for `<prog>`, Fig. 2.1). The operator is denoted by a pair of square brackets `[]`; the cells matching the constraints listed inside the brackets are *children of* the cells matched by the constraint outside and to the left of the brackets. When separated by commas, multiple child subprograms are allowed inside brackets. Each child subprogram therefore defines a spatial constraint with the same parent.

2.2.2.3 Geometric Descriptors

Geometric descriptors may be appended with a constant *quantity* (`<quant>`, Fig. 2.1), for example, `r{5}`, indicating that the child cell should be 5 cells to the right of the parent cell. In some cases, we need to express that a child cell has a

spatial relationship of indeterminate distance from its parent (e.g., "somewhere to the right"), and Kleene stars can be used for this purpose. As with many regular expression implementations, the Kleene star can be interpreted in several different ways. The language supports match-all, greedy, and non-greedy semantics using `*`, `*#`, and `*?` respectively.

2.3 Algorithm

Layout transformation synthesis *automatically generates* a FLARE program as output given a spreadsheet and a set of positive and negative examples of tuples as input. The synthesized FLARE program is guaranteed to extract all of the positive tuple examples and none of the negative tuple examples supplied by the user. We frame the problem of finding a satisfactory program as a *search* over all valid combinations of constraints that satisfy the positive and negative examples. Algorithm pseudocode is shown in Figs. A.3, A.4, A.5, and A.6 in the Appendix.

Let P be the set of user-provided tuples representative of the desired relational table, otherwise known as *positive examples*. Let N be the set of user-provided tuples representing counterexamples, otherwise known as *negative examples*. The space of all possible transformation programs over the columns in P and N is a graph called the *column graph*. Let each column in the output table be a vertex in this graph. Let constraints be directed edges between vertices; each edge encodes a (spatial constraint, cell constraint) pair. Since spatial constraints relate columns, edges are directed. Cell constraints define valid strings in the target column.

In order to correctly perform a transformation, a valid FLARE program must 1) refer to every column in the output table, and 2) define functional spatial dependence for every column, except the first column, which is the "root". This resulting structure is a spanning tree over the column graph.

The spanning tree search must also satisfy the examples in P and N. The fact that FLARE programs have a tree structure means that a blind, combinatorial search can be avoided. Instead, the program search procedure, shown in Fig. A.4, is an adaptation of Kruskal's minimum-weight spanning tree algorithm [61].

By construction, all programs in the search space satisfy the positive examples, but not the negative examples. To quickly find a program that satisfies both, heuristics assign weights to edges such that the user's *intended* program is *likely* to be a minimum-weight spanning tree. When this intuition is correct, greedy search pays off. When it is not, the search must backtrack. Heuristics are described in Chapter 2.4.

There may be many spanning trees that satisfy the examples given by the user. While all of these programs are *correct with respect to the user's examples*, not all of them are what the user *wants*. Inferring user intent from incomplete specifications is a difficult problem. Our heuristics also attempt to guide the search toward constraints most likely intended by users, reducing the number of examples needed (see Chapter 2.4).

2.3.1 Synthesis Goals

Informally, the synthesizer must perform the following tasks, given P and N: 1) For column each i in the output table, learn the set of all cell constraints that satisfy P (Fig. A.5). 2) For column pair i, j, learn the set of all spatial constraints that satisfy the user's *positive examples* (Fig. A.6). 3) Any combination of learned cell and spatial constraints that form a spanning tree T over all columns satisfies the user's positive examples, but many of them include tuples in N. The last step is to exclude any T that includes any member of N (Fig. A.4).

2.4 FlashRelate Implementation

The efficient operation of the synthesis algorithm depends on three key implementation ideas. *Ranking schemes* prioritize the search for likely good constraints over likely bad constraints. *Pruning* reduces the size of the search space, making search more efficient. *Compact data structures* based on bitvectors ensure that search uses little memory and logical operations are performed efficiently.

2.4.0.1 Ranking

At every step in the search, FlashRelate must choose between candidate constraints. Some constraints are more general while others are more specific. This choice impacts both the speed of the synthesizer and the number of examples required by the user. Favoring specific constraints may make the search fast, because they are more likely to rule out negative examples. Specific constraints may also mean that the user must provide more positive examples before the correct program is found. Conversely, favoring general constraints may fail to exclude negative examples, causing the search to backtrack frequently, slowing search. General constraints may require a user to provide more negative examples.

FlashRelate has a strong bias toward specificity. This enables users to focus on what they want, instead of what they don't want, which is a more natural form of interaction. This choice also allows for faster search. Constraints are ranked by the following heuristics, in this order: 1) **H1** Excluding more negative examples is favored over excluding few. 2) **H2** Specific spatial constraints are favored over general ones. This implies that multiple positive examples are required to learn non-constant-length spatial constraints. 3) **H3** Simpler geometric layouts (i.e, fewer "turns") are favored over complex ones. 4) **H4** Cells above and to the left of positive examples are implicit negative examples. Users typically visually scan a spreadsheet starting at the top left,

moving rightward and downward. Implicit negative examples are abandoned whenever they contradict positive examples.

2.4.0.2 Pruning

The naive approach of enumerating all possible geometric descriptors fails when one considers that there are an infinite number of possible constant-length descriptors. Instead, the enumeration phase of the search returns the smallest possible set of descriptors consistent with all the positive examples. Many descriptors can be ruled out based observations of user examples. For example, if in two positive examples, two cells belonging to the same column share a single candidate parent cell, constant-amount geometric descriptors are not possible and can be pruned.

2.4.0.3 Data Structures

When interpreted over a spreadsheet a cell constraint evaluates to a set of cells. This set of matching cells is represented as a bit vector, mapping a linear traversal of the spreadsheet to "1" (match) or "0" (no match) to bitvector indices. This representation is efficient for three reasons: 1) Match results are stored efficiently, and can also be cached efficiently, since many programs share subprograms that do not need to be re-evaluated. 2) Satisfiability can be answered using bit vector arithmetic. 3) Operations requiring bit-counting, which is used heavily in our heuristics, is $O(\#$ bits set) [90].

2.5 Evaluation

This section evaluates the design of Flare and FlashRelate on a variety of real-world spreadsheets. We answer the following questions: 1) It is possible to manually write Flare programs to perform a diverse set of extraction tasks? 2) Can FlashRelate automatically infer equivalent programs to the hand-written programs? 3) How effective are heuristics at reducing the time and number of examples required by the synthesizer?

2.5.0.1 Benchmark Selection

The evaluation considers two sets of benchmarks. The first set of benchmarks were borrowed from related work [50] that examined 51 table-transformation programs from Excel help forums. Nearly half (22) of the transformations were straightforward relational extraction tasks that could be expressed in FLARE. The remainder performed computation (e.g., arithmetic) outside the scope of extraction programs. A second set of benchmarks rounds out the evaluation with more difficult tasks selected from the EUSES corpus [43]. This second set of benchmarks was chosen specifically to test more challenging extraction tasks.

2.5.0.2 Expressiveness

To evaluate the expressiveness of FLARE, we manually wrote a correct FLARE program for each benchmark and extracted the resulting output table. The FLARE language is expressive enough to extract the desired tuples from all of the multi-dimensional data patterns we observed. We conclude that FLARE is an effective tool for performing extraction tasks.

2.5.0.3 Synthesizer Experiments

The following method is intended to simulate a user interacting with FLASHRELATE. The relational table extracted by the handwritten (i.e., "ground truth") program represents the user's desired extraction output; call this program the *oracle*. After each invocation of the FLASHRELATE algorithm by the simulated user with a set of examples, the synthesized program is considered correct if its output agrees with the oracle. When the FLASHRELATE output differs, the simulated user finds the first tuple that deviates from the oracle by scanning the extracted table from top to bottom. Deviant tuples come in two forms: 1) if a tuple from the oracle's output is missing from the program output, it is a positive example; 2) if a tuple from the program output does not appear in the oracle's output, it is a negative example. This process

is repeated until either the synthesizer finds a program whose output matches the oracle's or it times out. 10 minutes was chosen as the maximum total duration of the task as we felt few users would wait longer.

Six algorithm configurations were measured to understand the benefit of ranking choices. In all cases, regular expressions come from a small corpus (< 100) of common patterns combined with a simple regular expression generator. Each configuration examines the effect of adding a heuristic to the FlashRelate algorithm. Experiment configurations are: 1) **H1, H2, H3, H4**, 2) **H1, H2, and H3**, 3) **H1 and H2**, 4) **H1**, 5) **no ranking**, and 6) **R**. **H#** refers to the heuristic described in Chapter 2.4.

Configuration **R** changes the oracle model, choosing examples randomly. **R** tests whether FlashRelate is robust to a wide variety of user example-selection behaviors. Example order may matter because FlashRelate learns geometric differences from one example to the next. Anecdotally, examples located near each other in a spreadsheet are geometrically more similar than examples further from each other. When examples are chosen randomly, they are unlikely to be repeatedly drawn from the same neighborhood. The configuration enables all heuristics, runs the experiment 30 times, and averages the results.

2.5.1 Results

In the best case, the algorithm nearly always finds a solution within 10 minutes (Fig. 2.3). More than 80% of the benchmarks completed in less than 10 seconds *total*. Users typically wait only 1.6 seconds per iteration (median: 0.6 seconds). Users also need to provide only a few examples, for an average of 3.5 positive examples (median: 3 positive examples) and 2.0 negative examples (median: 1 negative example) (see Fig. 2.4b).

Ranking heuristics have a sigificant effect on speed, as with ranking, the algorithm finds a solution before the timeout 13x more often. Without ranking, the algorithm

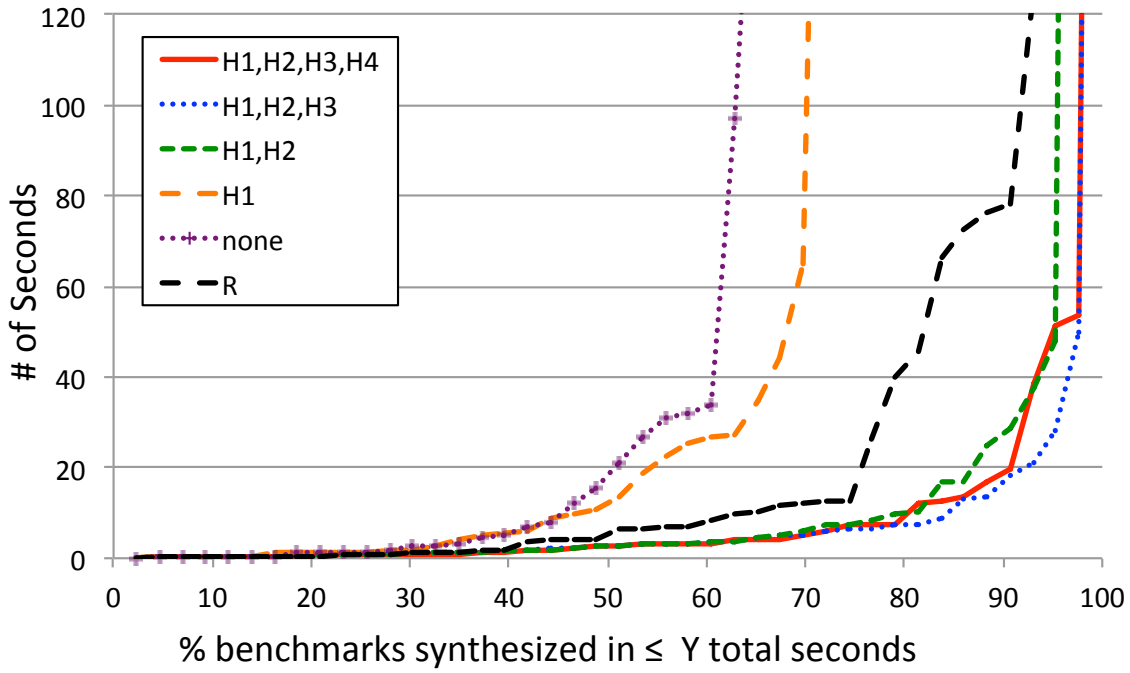


Figure 2.3: FlashRelate synthesis run time. Fewer seconds and a greater percentage is better (bottom right). FlashRelate does best with most heuristics enabled (**H1,H2,H3** and **H1,H2,H3,H4**). For example, **none** (no heuristics) succeeded on only 60% of the benchmarks before the timeout whereas **H1,H2,H3,H4** succeeded on all but one. **R** is a variation of **H1,H2,H3,H4** where the order of examples is randomized.

requires many more examples: an average of 3.0 positive and 13.7 negative examples (medians: 3 positive; 9 negative). Random example selection (config. **R** in Figures 2.3 & 2.3) does have a small, negative effect on the speed and number examples needed by the synthesizer, but heuristics still perform well even in this case. We think users are unlikely to choose examples randomly.

2.6 Conclusion

We present Flare, a formal language that expresses extraction queries against spreadsheets. We also present a layout transformation algorithm that synthesizes Flare programs from user-provided examples. Finally, we present FlashRelate, a layout transformation plugin for Microsoft Excel. We designed FlashRelate's

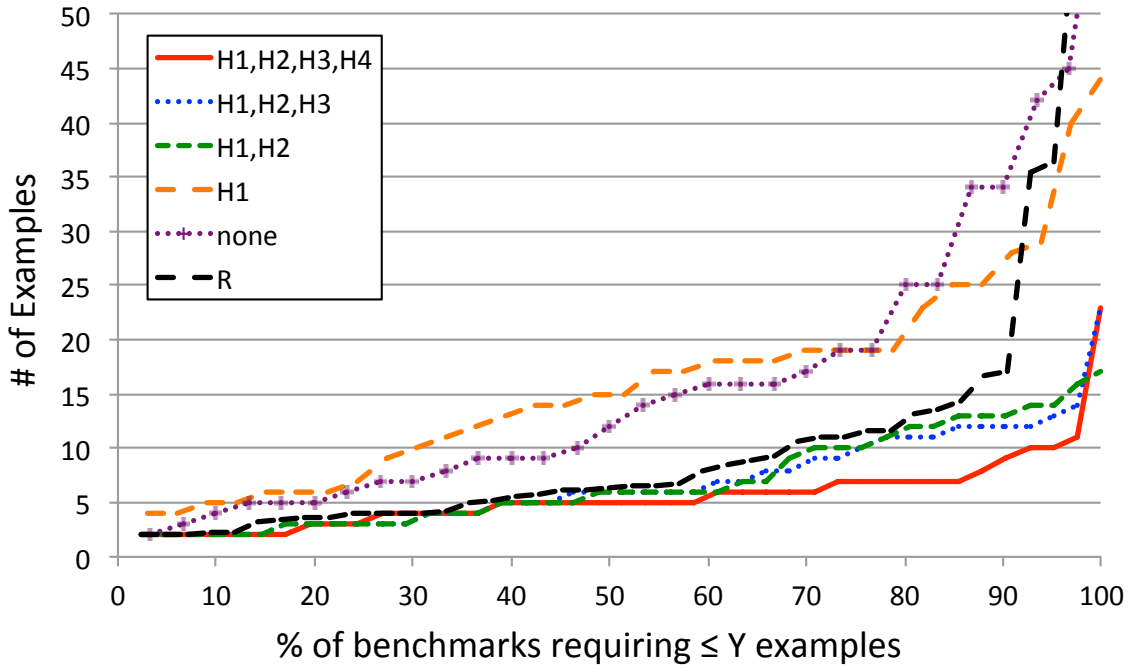


Figure 2.4: Number of examples needed for FlashRelate. Fewer examples and a greater percentage is better (bottom right). Generally, **H1,H2,H3,H4** requires fewer examples than the other configurations. **R** is a variation of **H1,H2,H3,H4** where examples are chosen randomly.

interface to be simple, fast, and efficient. Users need only point and click to obtain the extractions that they want. Notably, no programming is required to "unlock" data from ad hoc layouts.

CHAPTER 3

CHECKCELL: DATA DEBUGGING FOR SPREADSHEETS

The second challenge in data analysis is dealing with imperfect data. Data is routinely corrupted before it reaches a data analyst [51]. This chapter introduces *data debugging*[1], an approach that combines program analysis and statistical analysis to *automatically* find potential data errors.

Since it is impossible to know a priori whether data are erroneous or not, data debugging does the next best thing, locating input data that have an *an unusual impact on the computation*. Intuitively, inputs that have an inordinate impact on the final result are either very important or wrong. By contrast, wrong data whose presence have little or no impact on the final result do not merit special attention.

CheckCell is a data debugging tool designed as an add-in for Microsoft Excel and for Google Spreadsheets (Figure 3.2). CheckCell is best suited for large spreadsheets where manual auditing is onerous and error-prone. CheckCell highlights all inputs whose presence causes function outputs to be dramatically different than the function output were those outputs excluded. CheckCell guides the user through an audit one cell at a time, visiting the most severe likely errors first.

3.1 Approach

Data debugging is a hybrid program analysis and statistical technique that requires only the inputs and formulas present in a user's spreadsheet. Unlike typical machine

[1]This work appeared at OOPSLA 2014 [11].

	A	B	C	D	E
1	Assignment	Grade		*Homework*	20%
2	HW 1	84		*Quizzes*	30%
3	HW 2	77		*Exams*	50%
4	HW 3	92			
5	HW 4	93		**Final grade**	84.275
6	Quiz 1	87		**Pass/Fail**	Fail
7	Quiz 2	90			
8	Quiz 3	85			
9	Quiz 4	91			
10	Exam 1	84			
11	Exam 2	78			

Figure 3.1: A typical gradesheet containing an error. The formula in `E6` is `IF(E5 > 85, "Pass", "Fail")`. The transposition typo in B11 changes this student's grade from passing to failing. Gaussian outlier analysis fails to detect this error, but CheckCell does.

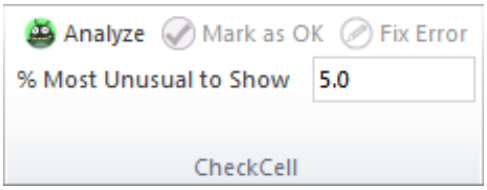


Figure 3.2: CheckCell only requires that a user specify the maximum percentage of spreadsheet inputs to audit. It then guides a user through an audit of highest-ranked error suspects.

learning approaches, data debugging does not make use of training data or weights. We begin by presenting an example, a typical gradesheet, in order to illustrate scenarios where the technique is useful.

3.1.1 Example

Consider the example spreadsheet in Figure 3.1, a typical grade sheet for a university course. Grade averages for different activities (homework, quizzes, exams) are weighted according a table and then summed to obtain a final grade. Finally, if the grade crosses a threshold (in this case, `85`), then the student is considered to have passed the course.

In this example, the error is a transposition of the value in cell `B11` from an `87` to a `78`. Since this grade is an exam, it is weighted more heavily than the grades for homework and quizzes. Note that a two-sided parametric outlier test based on the Gaussian distribution ($\alpha = 0.05$, two standard deviations) does not find this error. Grades are often normally distributed, so the Gaussian distribution appears to be an appropriate fit. A test using the Gaussian distribution will not find the error, however, because it is not an extreme value. The most extreme values are the values in cells `B3` (`77`) and `B5` (`93`). `78` is not just a valid grade, but in general, a common one. Nonetheless, this error changes this student's final outcome from `Pass` to `Fail`.

Data debugging is designed to find precisely this kind of subtle error. Using our implementation of data debugging, CheckCell, the user must first decide $k\%$, the proportion of input values that they want to inspect ("% Most Unusual to Show"). By default, this value is set to 5%, which is based on our empirical observation that users tend to mistype strings at this rate (See Chapter 2.5). After clicking the "Analyze" button, CheckCell computes likely errors and ranks them by their hypothesized severity.

Each error is presented to the user one-at-a-time. Upon being presented an error, the user must either mark the cell as correct ("Mark as OK") or fix the error ("Fix Error"). The auditing procedure terminates when either the user has examined at most $k\%$ of the inputs, or when CheckCell determines that none of the remaining inputs are likely errors, whichever is smaller. By increasing $k\%$, users may increase accuracy for a greater expenditure in effort. For this example, after a single iteration CheckCell finds only this single error, then it terminates.

3.2 Algorithms

This section describes data debugging in detail. Chapter 3.3 includes a formal analysis of its asymptotic performance and statistical effectiveness.

3.2.1 Dependence Analysis

Data debugging's statistical analysis is guided by the structure of the program present in a worksheet. The first step is to identify the inputs and outputs of those computations. A spreadsheet is scanned and all formula strings collected. Formulas are parsed using an Excel grammar. References to input vectors and other formulas are found using each formula's syntax tree. References to local, cross-worksheet, and cross-workbook cells are all resolved. Data debugging uses techniques similar to past work to identify dependencies in spreadsheets [41].

Data debugging's statistical analysis depends on the ability of the analysis to identify replacement input values when considering a suspected input error. When a function has only a scalar argument, namely a single cell or a constant, data debugging does not have enough information to reliably generate other representative values. Therefore, data debugging limits its analysis to vector inputs.

3.2.2 Impact Analysis

Data debugging operates under the premise that the value of a function changes significantly when an erroneous input value is *corrected*. More precisely, the analysis poses the (null) hypothesis that the removal of a value will *not* cause a large change in function output. The analysis then gathers statistical evidence in an attempt to reject this hypothesis.

Removing an input value requires replacing it with another representative value. Since the analysis never knows the true value of the erroneous input, it must choose from among the only other replacement candidates it can justify, namely other values in the same input vector as the suspected outlier.

3.2.2.1 Function Classes

We limit the analysis to formula inputs that are justifiably homogeneous, i.e., input values for order-independent vector functions. In an analysis of frequently-

used functions, we found that most spreadsheets in the EUSES corpus satisfy this requirement.

3.2.2.2 Non-Parametric Methods: The Bootstrap

Standard approaches to outlier rejection generally depend on the shape of the distribution. These so-called *parametric* methods require analysts to supply a distribution and its parameters. The normal distribution is commonly assumed. Given that the analysis needs to perform statistical tests on *any* formula over *unknown* data distributions, parametric methods cannot be justified.

Instead, input analysis incorporates an adaptation of Efron's bootstrap procedure, a *non-parametric* (distribution-free) statistical method [32]. We use the bootstrap to estimate the distribution of a function output, given only a sample of the distribution of inputs. With this, one can estimate the variability of the formula in question, allowing for reliable inference even when the sample size is small (*i.e.*, under 30 elements), or the distribution is unknown or difficult to compute.

The procedure works as follows: 1) Draw a random sample with replacement, $X_i = (x_0, \ldots, x_{m-1})$, from the input vector of interest. m is the size of the original sample. 2) Compute the function output for sample i, namely $\hat{\theta}_i(X_i)$. 3) Repeat this process n times (see Chapter 3.3.1).

The resulting distribution $\hat{\theta} = (\hat{\theta}_i, \ldots, \hat{\theta}_{n-1})$ gives an approximation of θ, the true distribution of function outputs. $\hat{\theta}$ can be used for inference, because the bootstrap procedure gives an indication of the variability of θ, i.e., we know which values of θ are unlikely.

3.2.2.2.1 Hypothesis test. To determine whether an input, x, is likely to be an error, the analysis conditions the output distribution $\hat{\theta}$ on the absence of x in the data. We call this conditional distribution $\hat{\theta}_e$. The conditional distribution approximates the effect of correcting the input error. If the original function output, θ_{orig}, is highly

unusual when compared to the $\hat{\theta}_e$, the input x is either a very important input or wrong. The analysis performs one of two variants of the hypothesis test, depending on whether a function is numeric or string-valued.

3.2.2.2.2 Numeric-output functions. For numeric outputs, the bootstrap distribution is sorted in ascending order, and the quantile function is applied to determine the confidence bound of interest. If θ_{orig} falls to the left of the $\alpha/2$th percentile or to the right of the $1-\alpha/2$th percentile of $\hat{\theta}_e$, we reject declare x an outlier.

3.2.2.2.3 String-valued-output functions. For string-valued function outputs, the bootstrap distribution becomes a multinomial. The multinomial is parameterized by a vector of probabilities, $p_0, \ldots, p_{k-1}$, where k is the number of output categories (in our case, distinct strings), and where $\sum_{i=0}^{k-1} p_i = 1$. The analysis calculates p_i from the observed frequency of category i from $\hat{\theta}_e$. If the probability of observing the original function output, θ_{orig}, is less than α, we declare x an outlier.

3.2.3 Impact Scoring

Finally, all inputs that failed at least one hypothesis test are ranked and presented to the user. There are $O(i \cdot f)$ hypothesis tests, one for each (input,output) pair, where i is the total number of inputs in the spreadsheet and f is the number of function outputs. Likely anomalies are highlighted in varying shades of red, with bright red cells being highly-ranked potential errors.

The analysis cannot know a priori which function outputs are the most important to the end-users. Arguably, inputs that have large effects on large-scale computations are more important than inputs that have large effects on small-scale computations. We utilize a weighing scheme to differentiate the two. The *total impact* of an error is defined as $\sum_{i,f} s_{i,f} \cdot w_f$ where $s_{i,f}$ is the impact score for input i and function f, and

where w_f is the weight of function f determined by the number of transitive inputs for that function.

3.3 Analysis

This section presents an analysis of data debugging's dominant contributor to the cost of accurate inference: the number of resamples required to perform the bootstrapping method. A mechanism for mitigating this cost is also discussed.

3.3.1 Number of Resamples

For an input vector of length m and a given value from that vector, x, the probability of randomly selecting a value that is not x is $\frac{m-1}{m}$. The probability of selecting m such values is therefore $(\frac{m-1}{m})^m$. As m grows, we obtain the following identity (see Appendix B for the proof):

Theorem 3.3.1.

$$\lim_{m\to\infty}(\frac{m-1}{m})^m = \frac{1}{e}$$

Statistical literature suggests that the number of bootstraps be at least 1000 when the computational cost is tolerable. For efficiency, we perform our bootstrapping procedure once for each input range, and then partition the resulting $\hat{\theta}$ distributions according to the value x of interest. We set $n = 1000 \cdot e$. Theorem 3.3.1 ensures that, on average, there are 1000 resamples in the bootstrap distribution for $\hat{\theta}_e$.

For i input ranges and a bootstrap size of n, data debugging requires $O(i \cdot n)$ time to analyze a spreadsheet. In practice, the caching feature described in the next section makes observing even this modest linear cost unlikely.

3.3.2 Efficiency of Caching

Data debugging's run time is $O(i \cdot n)$, or linear in the number of recalculations required, where i is the number of input vectors and n is the number of bootstraps

required. As n grows larger than m, the length of an input vector, the probability that a given resample will appear again during bootstrapping increases substantially. Our implementation of data debugging, CheckCell, caches the output of functions whose input values have been previously calculated.

For an input vector of length m and a resample X, it must be the case that the sum of the fingerprint counter's values equals m. There are only $f = \binom{2m-1}{m}$ ways to sum to m for a fingerprint vector of length m. There are only f possible fingerprints for an input vector of length m. Because input vectors are resampled uniformly at random, the probability of choosing a particular fingerprint is $\frac{1}{f}$. We expect to see a particular fingerprint with a frequency of $\frac{n}{f}$ for a bootstrap of size n. Clearly, for $n > f$, we are likely to observe a repeated fingerprint. As n grows larger than f in the limit, observing a repeated fingerprint is guaranteed.

3.4 CheckCell Implementation

CheckCell is written in a mixture of C# and F# for the .NET managed language runtime, written as a plugin for Microsoft Excel using the Visual Studio Tools for Office (VSTO) interoperability framework. It is compatible with Excel versions 2010-2016.

Unfortunately, no sufficiently accurate parser for Excel was freely available to use for this work (Microsoft's parser is not available to the public). Likewise, no Excel dependence analysis tool is available to the public. Consequently, we wrote two utilities to fill this gap, an Excel formula parser[2] and an Excel data dependence analysis engine[3].

[2]https://github.com/plasma-umass/parcel

[3]https://github.com/dbarowy/Depends

3.5 Evaluation

CheckCell was evaluated across three dimensions: its ability to reduce input errors, its ability to reduce end-user effort in fixing errors, and its execution time.

3.5.0.1 Experimental Methodology

CheckCell was run on a random selection of 61 benchmarks from the EUSES spreadsheet corpus. For each spreadsheet, a cell was randomly selected and its value perturbed with a representative error drawn from an error generator (see Sec 3.5.0.5). A simulated user examined and flagged cells as prompted by CheckCell. When the simulated user found a real error, the cell was marked as a true positive and corrected. When a cell did not contain an error, it was marked as a false positive. Remaining errors not flagged by CheckCell were considered false negatives. We repeated the procedure 100 times for each spreadsheet.

3.5.0.1.1 Baselines. The baseline for CheckCell's effort reduction is a cell-by-cell manual audit. Nonetheless, an experienced analyst might expect standard outlier detection techniques to be useful, so CheckCell is also compared against a Gaussian outlier detection method (referred to as `NAll`). We report CheckCell's results with the *% Most Unusual to Show* threshold set at 10% (`CC10`). Note that this setting means that CheckCell may report *up to* 10% of the values in the spreadsheet. In practice, this rarely occurs.

3.5.0.1.2 Procedure. The evaluation introduces a single outlier into each spreadsheet (i.e., there is at most one true positive). While input perturbations are drawn from a typo model, no effort is made to ensure that such errors are *important*. Our experimental design lets one compare the sensitivity of the two different techniques across two dimensions: (1) the magnitude of the input error, and (2) the magnitude of the output error. Finally, to simplify the comparison, experiments were limited

numerical functions. The limitation biases the experiment in favor of `NAll`, since CheckCell is strictly more powerful, but it makes the comparison straightforward.

3.5.0.1.3 Latent Errors. Evidence suggests that users make input errors at a rate of roughly 5% per string (see Chapter 3.5.0.5). Therefore, it is likely that these spreadsheets already contain errors. We conservatively assume that all flagged cells without ground truth are false positives. These experiments likely underreport CheckCell's precision.

3.5.0.2 Quantifying User Effort

Without an auditing tool, users must in the worst case inspect all function inputs. An effective tool should reduce the number of inputs a user must manually examine. Let z be the number of cells inspected during the use of the tool ($z \leq m$, the total number of inputs). The *relative effort* of the tool is then defined as effort $= z/m$.

3.5.0.3 Quantifying Error

Measuring the magnitude of an input perturbation is straightfoward. Measuring the magnitude of a spreadsheet's change in outputs needs to account for the fact that even simple spreadsheets often have multiple outputs. The *total output error* metric measures the magnitude of an output change relative to other outputs.

Let the "correct" (original) spreadsheet be a vector S of strings. Let the error-injected spreadsheet be a vector S_e of strings. Let the "corrected" spreadsheet after an outlier procedure is run be S_c. Let f be a real-valued function over a subset of spreadsheet inputs (a spreadsheet formula). Then the *normalized error* of f is:

$$\text{err}(f) = \frac{|f(S_c) - f(S)|}{|f(S_e) - f(S)|}$$

We compute the *total error* of a spreadsheet as follows. Let the set of all numeric functions defined in a spreadsheet be F. The total error after correction is:

$$\text{err}_{tot} = \sum_{f \in F} \text{err}(f)$$

3.5.0.4 Classifier Accuracy

CheckCell's aims to assist a user in a spreadsheet audit by classifying inputs into one of two categories: errors and non-errors. CheckCell cannot distinguish between important errors and important non-errors. Nonetheless, it is informative to examine CheckCell's error-finding accuracy using off-the-shelf classifier metrics. We use *precision* and *recall* to serve this purpose.

3.5.0.5 Error Generator

In order to inject representative errors into spreadsheets, we built and trained an error model using Amazon's Mechanical Turk to perform data entry tasks. The model is designed to generate two kinds of errors: (1) character transpositions and (2) simple typographical errors. Input data came from two sources: A random sampling of formula inputs from 500 spreadsheets in the EUSES corpus (corresponding to 69,112 input strings) and 100,000 randomly generated strings. Overall, 5.26% of all retyped strings exhibited at least one typo, consistent with similar studies [77].

3.5.1 Experimental Results

The distribution of perturbations and their effects over 2836 experiments is shown in Fig. 3.3. Input errors are nearly Gaussian (quantiles: 0% = -14.41, 25% = -0.04, 50% =-0.04, 75% = 0.30, 100% = 13.69). Output error is skewed; small errors dominate (quantiles: 0% = 0.00, 25% = 0.03, 50% = 0.08, 75% = 0.25, 100% = 1.00).

3.5.1.1 Precision and Recall

Across all benchmark runs, CheckCell had a mean precision of 8.0% and a mean recall of 12.1%. `NAll` had a mean precision of 5.9% and a mean recall of 15.8%. A

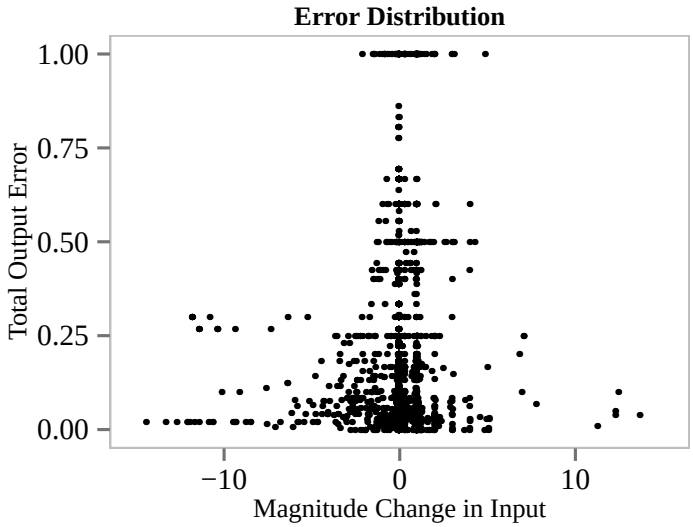
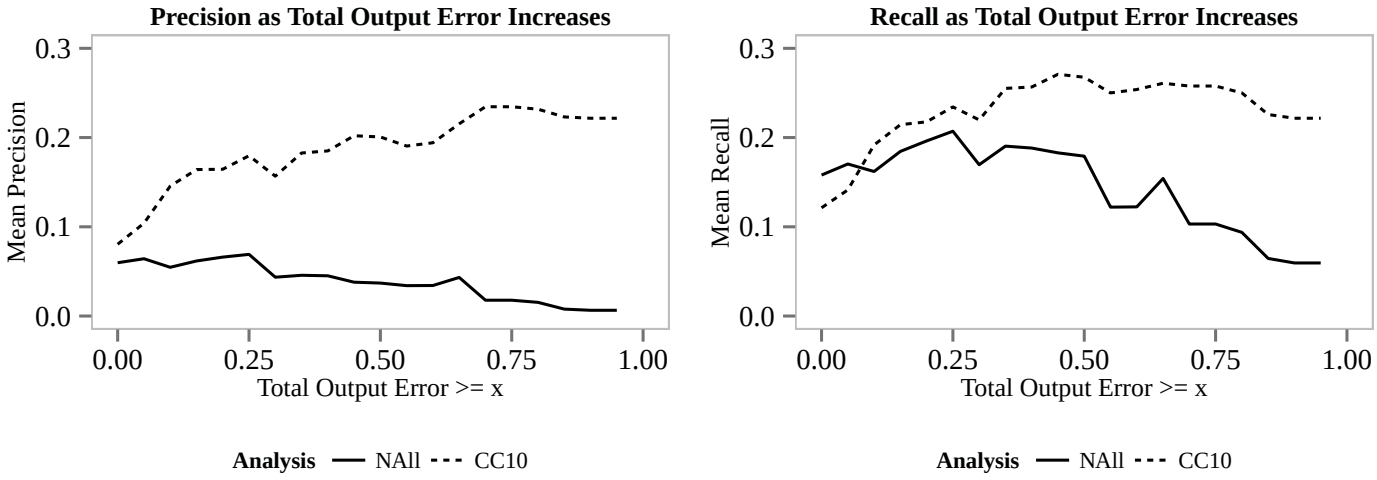


Figure 3.3: The distribution of input errors. Each point corresponds to a single benchmark run. The change in input magnitude as the result of the error is shown on the x-axis while the change in the spreadsheet's total error is shown on the y-axis. Note that because the typo generator is designed to produce representative errors, it is biased toward small-magnitude perturbations that don't matter.



(a) (b)

Figure 3.4: Precision and recall. (a) Precision as the minimum total output error is increased. CheckCell always has fewer false positives than `NAll`. (b) Recall as the minimum total output error is increased. For errors that cause a small effect, `NAll` returns more false positives, but as errors grow more severe, CheckCell returns increasingly relevant errors.

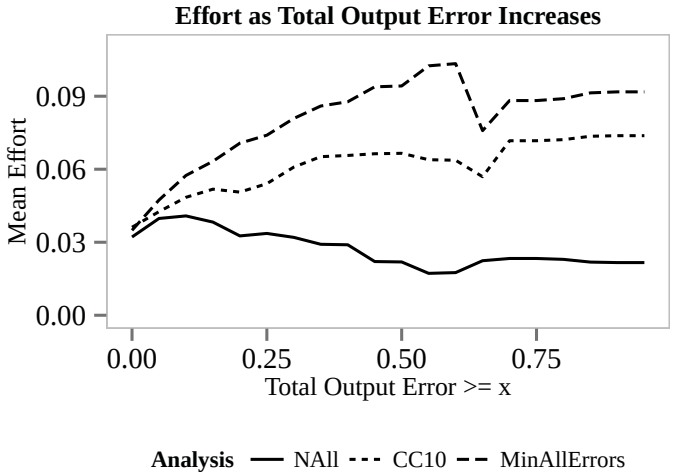


Figure 3.5: For errors that cause a small total error, CheckCell requires about the same mean effort as `NAll`.

random-answering adversary that expects errors to occur at a rate of 5.26% has a mean expected precision of 3.5% and a mean expected recall of 5.26%. CheckCell has higher precision than `NAll`, indicating that it is more discriminating. However, `NAll` has a higher recall, which means it flags more errors than CheckCell. Nonetheless, both of these figures are strongly influenced by the presence of a large number of small errors with little impact. The skew is an artifact of the error generator, since most errors generated were small and had little impact.

Precision and recall numbers are more informative when one stratifies benchmarks by a minimum total output error. Figure 3.4(a) compares `CC10` and `NAll`. Figure 3.4(b) compares `CC10` and `NAll` mean recall. `CC10` gains a rapid precision advantage over `NAll` as errors have more of an effect on the computation. `NAll`'s initial recall advantage over CheckCell also evaporates as errors grow in importance. As errors grow in importance, CheckCell finds them more accurately than `NAll`.

3.5.1.2 Effort

CheckCell and `NAll` require comparable effort. Across all benchmarks, `CC10` required users to examine 3.6% of a spreadsheet's inputs while `NAll` required users to examine 3.2%. Again, as one stratifies by required effort, the picture changes. Figure 3.5 shows that for larger output errors, CheckCell requires users to inspect between 4% and 7% of the inputs. For the same errors, `NAll` typically requires users to inspect between 2% and 4% of the inputs. This odd effect is because `NAll` is *too* thrify: it frequently detects nothing at all, saving user effort only by missing *important* errors. In fact, `NAll`'s recall is only better when errors don't matter (Figure 3.4(b)). `NAll` is the most sensitive and requires the greatest user effort for the class of unimportant errors; CheckCell behaves in precisely the opposite manner.

3.5.1.3 Execution Time

The mean runtime over all spreadsheets is 6.42 seconds, with a median runtime of 2.98 seconds. For all but two of the 61 benchmarks, CheckCell typically takes 30 seconds or less to complete, and never takes more than 70 seconds. As the analysis in Chapter 3.3 predicts, cost is dominated by impact analysis, which is dependent on the number of inputs. Given the short execution times observed, we view this overhead as acceptable for an error detection tool.

3.5.1.4 Summary

CheckCell is more precise than outlier analysis and the errors found by CheckCell are more impactful. While CheckCell has lower recall than `NAll`, the errors missed by CheckCell are inconsequential. CheckCell is always more precise. Even when outlier analysis has the greatest possible advantage (numerical functions), CheckCell makes better use of a user's limited attention, and focuses user effort on the most important errors. Given CheckCell's support for a richer class of non-numeric functions, CheckCell is more useful across a wider range of spreadsheets.

3.5.2 Case Study: The Reinhart and Rogoff Spreadsheet

In 2010, the economists Carmen Reinhart and Kenneth Rogoff, both now at Harvard, presented results of an extensive study of the correlation between indebtedness (debt/GDP) and economic growth (the rate of change of GDP) in 44 countries and over a period of approximately 200 years [83, 84]. The authors argued that there was an apparent "tipping point": when indebtedness crossed 90%, growth rates plummeted. The results of this study were widely used by politicians to justify austerity measures taken to reduce debt loads in countries around the world [55].

Although Reinhart and Rogoff made the original data available that formed the basis of their study, they did not make public the instrument used to perform the actual analysis: an Excel spreadsheet. Herndon, Ash, and Pollin, economists at the University of Massachusetts Amherst, obtained the spreadsheet. They discovered several errors, including the apparently accidental omission of five countries in a range of formulas [55]. After correcting for these and other flaws in the spreadsheet, the results invalidate Reinhart-Rogoff's conclusion: no tipping point exists for economic growth as debt levels rise.

While some of the errors in the Reinhart-Rogoff spreadsheet are out of scope for CheckCell, we wanted to know whether CheckCell would be able to verify any of the other errors or discover new ones. We obtained the Excel spreadsheet directly from Carmen Reinhart and ran CheckCell on it. CheckCell singled out one cell in bright red, identifying it as a value with an extraordinary impact on the final result. We reported this finding to one of the UMass economists (Michael Ash). He confirmed that this value, a data entry of `10.2` for Norway, indicated a key methodological problem in the spreadsheet. The UMass economists found this flaw by careful manual auditing after their initial analysis of the spreadsheet (emphasis ours) [6]:

> For example, Norway spent only one year (1946) in the 60-90 percent public debt/GDP category over the total 130 years (1880-2009) that Norway appears in the data. Norway's economic growth in this one

> year was 10.2 percent. **This one extraordinary growth experience contributes fully 5.3 percent (1/19) of the weight for the mean GDP growth in this category even though it constitutes only 0.2 percent (1/445) of the country-years in this category.** Indeed Norway's one year in the 60-90 percent GDP category receives equal weight to, for example, Canada's 23 years in the category, Austria's 35, Italy's 39, and Spain's 47.

This case study demonstrates data debugging's utility not only for detecting errors but also for understanding structural flaws in computations.

3.6 Conclusion

This dissertation presents *data debugging*, an approach aimed at finding potential data errors by locating and ranking data items based on their overall impact on a computation. Intuitively, errors that have no impact do not pose a problem, while values that have an unusual impact on the overall computation are either very important or incorrect. We present the first data debugging tool, CheckCell, which operates on spreadsheets. We evaluate CheckCell's performance analytically and empirically, showing that it is reasonably efficient and effective at helping to find data errors.

CHAPTER 4

EXCELINT: DEBUGGING SPREADSHEETS WITH SPATIAL AND STRUCTURAL ANALYSIS

The last challenge in data analysis is writing correct analyses using spreadsheets. This dissertation presents *spatio-structural analysis*, a novel statistical and static analysis technique that automatically identifies errors in spreadsheets. Spatio-structural analysis reduces the problem of finding errors to that of finding anomalous formulas—both in terms of their structure and of their position on a spreadsheet—without relying on heuristics or domain knowledge. Previous work in conventional programming languages has shown that anomalous code is often wrong [25, 29, 34, 49, 82, 93]. Spatio-structural analysis extends this observation into the setting of spreadsheet formulas.

ExceLint[1] is the first spatio-structural analysis tool used to analyze spreadsheets, written as a plugin for Microsoft Excel. ExceLint has two novel visualizations designed to aid the programmer in finding and fixing formula errors. The first visualization—a kind of lightweight program understanding tool—overlays a spreadsheet with a *regularity map*. A regularity map shows programmers those regions of a spreadsheet where data reference invariants are maintained. This visualization helps users quickly locate errors by taking advantage of the innate human ability to perceive deviations from visual patterns: discrepancies "pop out", making them easy to find (see Chapter 4.3.1.1). The second mode is a visualization for *proposed fixes*

[1]Barowy, Daniel W., Berger, Emery D., and Zorn, Benjamin. ExceLint: Debugging Spreadsheets with Spatial and Structural Analysis. Manuscript in preparation.

that helps users quickly locate the most egregious errors and their suggested fixes, and is particularly useful for very large spreadsheets that are hard to view all at once (see Chapter 4.3.1.2). Both modes complement each other. For example, programmers can use the regularity map to understand the context for proposed fixes.

ExceLint is evaluated against 29 annotated worksheets drawn from the EUSES spreadsheet corpus [42]. The evaluation shows that the problem of finding reference errors in a spreadsheet can effectively be reduced to the statistical problem of finding anomalous references, without relying on heuristics or domain knowledge. ExceLint dramatically reduces the effort needed to audit spreadsheets and it improves on the state of the art by providing a high precision, user-friendly bug finding tool.

4.1 Approach

Spatio-structural analysis consists of three key phases. The first phase abstracts spreadsheet formula references, summarizing them only by their reference behavior. The second phase mines likely reference invariants, by clustering formulas by their fingerprints. The third phase uses this clustering to identify anomalous formulas and data. It then proposes "fixes," those likely errors and their corrections, which it ranks by their impact on an error model of the spreadsheet. Finally, depending on the visualization requested by the user, either the clustering or the ranked proposed fixes are returned and displayed.

4.1.0.1 1st Phase: Discovering Formula Reference Behavior

A *reference vector* is a novel spreadsheet formula abstraction that unifies spatial and structural information into a single geometric construct, namely a vector. Spatial information, namely the position of a formula within a spreadsheet, is encoded as the "tail" of a vector (i.e., the "origin"). Structural relationships, namely the locations

of data dependencies from the formula's data dependence graph, are encoded as the "head" of the vector (i.e., the "target").

Since spreadsheets are three-dimensional constructs—cells in a worksheet are located in a two dimensional grid, and spreadsheet workbooks often contain multiple worksheets—both the "tail" and the "head" of a reference vector are encoded using x, y, and z coordinates. The "head", which is the location of a referent, is denoted x', y', and z'.

For example, given a cell `C3` containing the formula `=A3 + B3 + 4`, which has three data references (`A3`, `B3`, and `4`), spatio-structural analysis produces three vectors, one for each reference: `C3`→`A3`, and `C3`→`B3`, and `C3`→`4`.

Constants embedded in a formula pose a challenge to represent in vector form, such as the reference vector `C3`→`4`. A naive vector encoding would produce zero-length vectors, since constants are co-located with the formula in the formula expression. To address this, reference vectors have an additional dimension, *dc*, which is 1 if there is an embedded constant, otherwise 0. Reference vectors are discussed in more detail in Chapter 4.2.3.

The set of reference vectors for `C3`→`A3`, and `C3`→`B3`, and `C3`→`4` are encoded as `<3,3,0,-2,0,0,0>`, `<3,3,0,-1,0,0,0>`, and `<3,3,0,0,0,0,1>`, respectively. Note that, by design, a different formula with a similar reference pattern yields a similar set of reference vectors. For example, the formula `=A4 + B4 + 5` located at cell `C4` yields the reference vectors `<3,4,0,-2,0,0,0>`, `<3,4,0,-1,0,0,0>`, and `<3,4,0,0,0,0,1>`.

Since later phases of spatio-structural analysis need to compare the reference behavior of two or more formulas, each formula's set of reference vectors is compressed into a *formula fingerprint* using a *vector hash function*. The precise vector hash function used depends on the phase of the analysis, but in general, fingerprints summarize reference vectors so that the degree of similarity between two formulas can be computed with an appropriate distance function. For example, the *location*

fingerprint for the formula `=A3 + B3 + 4` in cell `C3` is `<3,3,0,-3,0,0,1>`. The location fingerprint for the formula `=A4 + B4 + 4` in cell `C4` is `<3,4,0,-3,0,0,1>`. When paired with Euclidean distance, the distance between formulas `C3` and `C4`, which both perform the same computation in adjacent cells, is 1, the smallest distance between any two formulas on a spreadsheet. A *location-free fingerprint* elides location information such that both formulas in `C3` and `C4` yield the fingerprint `<-3,0,0,1>`; the Euclidean distance between them is zero. Zero distance between two fingerprints indicates that the two functions have the same reference behavior[2].

Vector hash functions and fingerprints are described in detail in Chapter 4.2.4.

4.1.0.2 2nd Phase: Finding Likely Reference Invariants

The second phase is to identify spatial and structural patterns using the vector fingerprint summaries described in the previous step. To achieve this, spatio-structural analysis performs a novel *rectangular clustering analysis* to identify reference patterns. Because rectilinear composition of spreadsheet formulas is encouraged by spreadsheet tools (see Chapter 1.1), the cluster analysis decomposes a spreadsheet into distinct rectangular regions characterized by the same reference pattern. Clusters indicate the programmer's likely intent to maintain a reference invariant in a given region of a spreadsheet. For example, if formulas in cells `C3` and `C4` described earlier were the only formulas in the spreadsheet, they would likely belong to the same cluster. Numeric data, headers, and whitespace are also clustered.

Clustering is described in detail in Chapter 4.2.5.

4.1.0.3 3rd Phase: Finding Likely Bugs

Finally, spatio-structural analysis uses the rectangular clustering to construct an error model. Because a reference error results in a formula with a different formula

[2]Except in the case of hash collisions, since fingerprints are actually hashes.

fingerprint than the intended formula, the clustering will reflect this fact; formula clusters never contain formulas with different location-free fingerprints.

We observe that a common mistake is failing to correctly update references for copy-and-pasted formulas. These erroneous formulas tend to produce an irregular clustering, since mistakes are adjacent to formulas that correctly maintain a reference invariant. An important property of the rectangular clustering is that irregularities produce more a more complex clustering—literally, more rectangles—and a more complex clustering results in an increase in entropy when compared to simpler clusterings. Spatio-structural analysis uses this difference in entropy to find likely reference bugs.

Because spatio-structural analysis utilizes an anomaly-based approach, suspected bugs are always paired with likely reference invariants. This pair is naturally suggestive of a *proposed fix*. A proposed fix models the effect of correcting a spreadsheet formula error by replacing the erroneous formula with the correct formula. A good fix reduces the model's entropy while a bad one either has no effect or increases it. Nonetheless, a simple reduction in entropy is not sufficient to find a good fix because of the presence of pathological fixes such as the following: replace all formulas with a single formula. The entropy of such a fix (ignoring other cells such as whitespace and data) will be zero. Instead, the model augments the fix fitness criterion with two other factors: the *fix distance* between a formula's and the invariant's references, and the *invariant significance*, literally, the number of nearby cells that observe the invariant.

Proposed fixes are ranked in a total order by the model, thresholded based on a user-definable effort parameter (by default, 5%, based on the prior likelihood of formula errors), and returned upon user request.

Proposed fixes and the error model are described in detail in Chapters 4.2.6 and 4.2.7.

4.1.1 Contributions

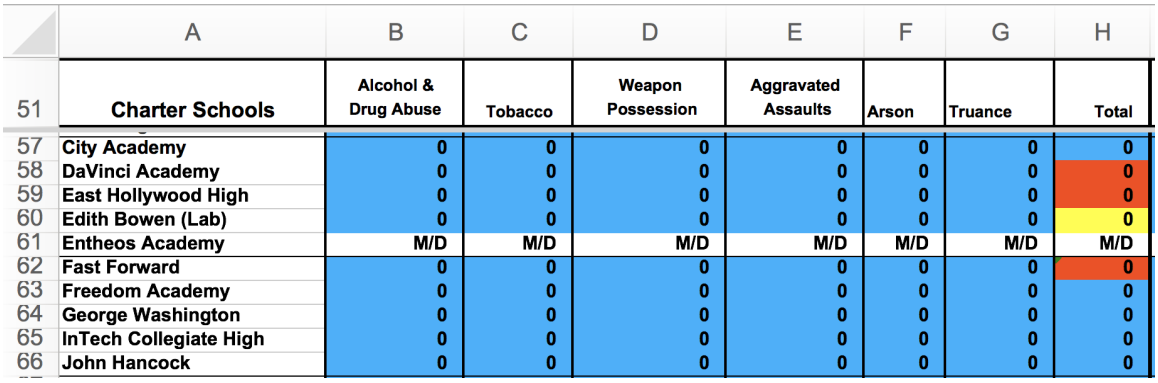
Spatio-structural analysis and the ExceLint tool address the challenge of finding errors in an increasingly important domain—spreadsheets—for an audience that has little programming ability—ordinary computer users. This dissertation chapter makes the following contributions:

- **Hybrid Statistical Static Analysis.** Spatio-structural analysis is a novel anomaly-based approach that casts likely program errors as a kind of statistical outlier. While spatio-structural analysis is designed for spreadsheet programs, it requires no large data sets, no model tuning, utilizes no heuristics, and little domain knowledge outside basic reasoning about tabular programming environments.

- **Suggests Fixes in Context.** Prior research into finding spreadsheet problems primarily focuses on "flagging" potentially bad values. Spatio-structural analysis goes a step further: potential errors are always returned with their suspected invariants.

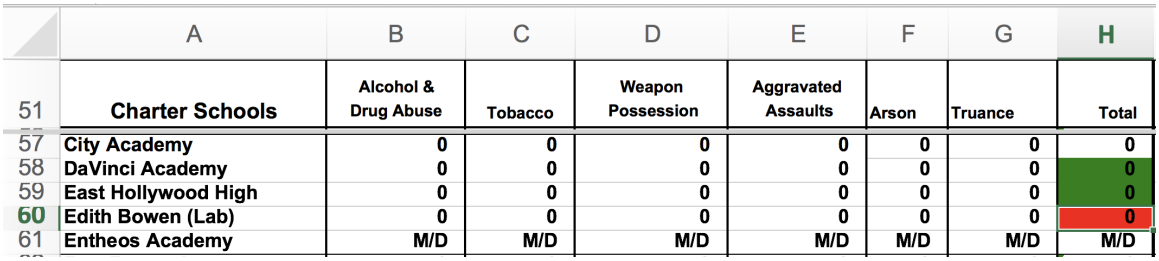
- **User-Friendly Visualizations.** ExceLint performs a complex hybrid statistical and static program analysis on user programs, but these technicalities are hidden from users. Instead, users interact with the analysis via a point-and-click interface and intuitive visualizations that make reference errors easy to see. These visualizations allow a user to understand why a potential fix is being suggested, without having to understand the deep data dependence relationships present in their spreadsheet.

- **Interactive-Level Performance.** Analyses must be carefully designed in order to be useful in an interactive environment such as a spreadsheet bug finder. The median runtime over a comprehensive benchmark suite shows that ExceLint runs in just 7.83 seconds.



	A	B	C	D	E	F	G	H
51	Charter Schools	Alcohol & Drug Abuse	Tobacco	Weapon Possession	Aggravated Assaults	Arson	Truance	Total
57	City Academy	0	0	0	0	0	0	0
58	DaVinci Academy	0	0	0	0	0	0	0
59	East Hollywood High	0	0	0	0	0	0	0
60	Edith Bowen (Lab)	0	0	0	0	0	0	0
61	Entheos Academy	M/D	M/D	M/D	M/D	M/D	M/D	M/D
62	Fast Forward	0	0	0	0	0	0	0
63	Freedom Academy	0	0	0	0	0	0	0
64	George Washington	0	0	0	0	0	0	0
65	InTech Collegiate High	0	0	0	0	0	0	0
66	John Hancock	0	0	0	0	0	0	0

Figure 4.1: Finding a bug using ExceLint's regularity map. The map shows that the programmer fails to maintain a reference invariant in column `H`. The user can immediately see that something is amiss without having to inspect every formula.



	A	B	C	D	E	F	G	H
51	Charter Schools	Alcohol & Drug Abuse	Tobacco	Weapon Possession	Aggravated Assaults	Arson	Truance	Total
57	City Academy	0	0	0	0	0	0	0
58	DaVinci Academy	0	0	0	0	0	0	0
59	East Hollywood High	0	0	0	0	0	0	0
60	Edith Bowen (Lab)	0	0	0	0	0	0	0
61	Entheos Academy	M/D	M/D	M/D	M/D	M/D	M/D	M/D

Figure 4.2: Finding a bug using ExceLint's proposed fix tool. The tool shows that `H60` is significantly unusual and suggests that it should be more like the cells in `H58` and `H59`.

4.1.2 Spatio-Structural Analysis

As with all anomaly-based analyses, it is important to find a domain where rare violations of a template model the occurrence of the error in the real world. In spreadsheets, while nearly all spreadsheets contain at least one error, the per-cell error rate is low, typically around 5% [11, 76]. Anomaly analysis has the potential to find errors in spreadsheets without having to manually extract domain rules.

We build on the anomaly analysis approach with *spatio-structural program analysis*. Spatio-structural analysis focuses on a single statistical feature: reference invariants. Reference invariants state that a given set of formulas must all obey the same reference behavior. Spatio-structural analysis goes beyond frequency counts of templates:

frequencies are conditioned by neighboring spreadsheet constructs and are checked against an error model before being ranked and presented to the user. Spatio-structural analysis is the first such anomaly analysis designed to find bugs in spreadsheet formulas, and requires no hard-coded rules of any kind.

Figure 4.1 shows ExceLint's *regularity map* visualization, which is built on top of spatio-structural analysis. It is immediately apparent that something is unusual with the cells in column `H`. First, cell `H57` is colored blue, which indicates that it is data like the cells found to its left. In fact, this cell should be a formula. Cells `H58`, `H59`, and `H62` stand out because they are colored orange. All of these cells exhibit an off-by-one reference error that instead computes the row total for the row one below. Finally, cell `H60` is colored yellow. This cell exhibits an off-by-two reference error that computes the row total for the row *two* cells below. When using the regularity map visualization, all of these problems immediately "pop out."

Figure 4.2 shows ExceLint's *proposed fixes* visualization. This visualization shows that cell `H60`, colored in red, is significantly different than other cells marked in green. The green color suggests that the formula in `H60` should be "more like" the formulas in `H58` and `H59`.

4.1.3 Reference Bugs

Since much prior work on bug-finding for spreadsheets focuses on either pattern-based or smell-based detection, authors generally avoid defining the performance of their tools in terms that state definitively whether a flagged cell or program construct is incorrect. While many of these tools produce helpful output, particularly when it comes to bad style, users cannot be certain that the tools actually help them find bugs.

This work attempts to find bugs, and defines its purpose narrowly. Specifically, spatio-structural analysis searches for *reference bugs*. Reference bugs violate reference

invariants. Specifically, they either (1) include an extra reference, (2) omit a reference, or (3) misreference data (e.g., they point to the wrong cell) when compared to other formulas in a computation. A reference bug does not a guarantee that the calculation returns the wrong value. For instance, accidentally summing over an additional cell that happens to be blank does not produce the wrong value. Nonetheless, we view the formula as wrong because a user who enters a value in the blank cell—thinking that the cell does not relate to the calculation—will introduce a calculation error into the spreadsheet.

As discussed in Chapter 1.3.1, reference bugs come in two varieties: *manifest errors* and *latent errors*. Manifest errors produce the wrong output. Latent errors may produce the wrong output if the user later modifies the spreadsheet. For example, accidentally summing over an extra cell will produce the wrong value if the user later enters a value into that cell without realizing that the sum refers to it. These errors are latent because future spreadsheet maintenance may cause them to become manifest errors.

Spatio-structural analysis fundamentally cannot determine whether a reference anomaly is a reference bug or simply an unusual but correct construction. Such judgements require understanding user intent, which is beyond the capabilities of any static analysis. Nonetheless, such anomalies are strongly suggestive of errors, as our evaluation shows (see Chapter 4.4).

4.1.3.1 Bug Duals

One consequence of anomaly detection is that it draws attention to both anomalous and non-anomalous code. While anomaly detection reveals many real bugs, it is not always the case that the true bug is the anomaly. Sometimes the true bug is the prevailing pattern while the anomaly is correct. In the context of spreadsheets, these kinds of errors can arise because copying and pasting of formulas is common. When a

user neglects to update references for the copy-and-pasted set of new formulas, the majority of the formulas can be wrong. Our experience with the ExceLint tool suggests that it is sufficient to flag either the anomaly or the non-anomaly in order to find the bug. In fact, this motivated the design of the formula fix tool which always highlights *both* anomalies and non-anomalies in a paired fashion during an audit (see Chapter 4.3.1.2).

We call these pairs *bug duals*. Formally, a bug dual is a pair containing two sets of cells, (c_1, c_2). In general, we do not know which set of cells, c_1 or c_2, correctly maintains an invariant. We do know, however, that all cells in c_1 maintain one invariant and that all cells in c_2 maintain another. As a denotational convention we always refer to the smaller set as *anomalous* and the larger set as *non-anomalous*. Not surprisingly, bug duals mirror the structure of proposed fixes (see Chapter 4.2.6); the presence of differing invariant clusters is, in fact, *how* spatio-structural analysis identifies candidate fixes.

4.2 Algorithms

Spatio-structural analysis performs a number of component analyses, which are described in the following sections.

4.2.1 Parsing and Data Dependence Analysis

ExceLint's analysis of a spreadsheet begins by *parsing* all spreadsheet formulas and running a *data dependence analysis*. Both analyses are frequently utilized as components in more complex analyses like the one performed by ExceLint. Since both techniques are well known, we only briefly describe them here.

4.2.1.1 Parsing

The first phase, parsing, converts the program text found in Excel spreadsheet formulas, such as `=SUM(A1:A10)` into an abstract syntax tree, a tree that represents a

set of program inputs and the operations to be performed on them. ExceLint uses the AST of formulas in order to understand the semantics of a given formula, it's dependence on other formulas and values, and its location in the spreadsheet.

4.2.1.2 Dependence Analysis

The second phase, a dependence analysis, is a well-known static analysis that determines whether a program statement depends on another program statement [26]. For example, the pseudocode program `a := 1; b := a + 1;` exhibits a data dependency from `b` to `a`, meaning that `b` cannot be computed without first computing `a`.

One form of a dependence analysis produces what is called a *data dependence graph*. A data dependence graph represents data dependencies as directed edges in a graph. Program statements are nodes in the graph. By building such a graph, an analysis has access to data dependence information for every statement in a program. While spreadsheets are strictly functional programs and do not have program statements as in traditional procedural programming languages, data dependence graphs can still be constructed for them.

Let G be the data dependence graph for a spreadsheet program. Let V be the set of cells in a spreadsheet, and let E be the set of data dependencies between cells. Since Excel is a *pure, functional programming language*, its dependence graph normally contains no cycles[3]. Note that we introduce cycles into this graph to denote embedded constants, which can be thought of a data references to the same location that a formula resides. These are the only cycles in what would ordinarily be a directed acyclic graph.

ExceLint utilizes the procedure in Figure 4.3 to obtain the dependence graph for a spreadsheet. Note that Excel distinguishes between *single-cell references*, which

[3]Cycles are exceedingly rare in real-world use, they not obvious to implement, and if you do happen to implement a cycle, Excel issues a warning. The semantics of cycles are of a *fixed point iteration* so that Excel programs always terminate.

point to a single cell, and *reference ranges*, which point to a (typically) contiguous, rectangular region of cells. Reference ranges are often used in formulas that take vectors inputs, such as `SUM`.

The time and space complexity of the dependence analysis are both proportional to the number of references found in a given workbook. In practice, the cost of parsing and building the graph is outweighed by the cost of marshaling spreadsheet data from Excel into the analysis. Marshaling is expensive because data objects that reside with Excel are written using the Component Object Model (COM) framework [68] and must be copied and converted from an Excel native-language process to an ExceLint managed-language process (see Chapter 4.3).

```
DEPENDENCE-ANALYSIS(V)
 1  E = {}
 2  for each cell v ∈ V
 3       if v is a formula
 4            ast = Parse(GetFormula(v))
 5            refs = ExtractReferences(ast)
 6            for each ref ∈ refs
 7                 if ref is a range
 8                      E = E ∪ ref
 9                 else E = E ∪ {ref}
10  return (V, E)
```

Figure 4.3: Spreadsheet dependence analysis algorithm. It takes in a set of cells, V, and returns a graph, $G = (V, E)$. The function `ExtractReferences` returns the set of edges between the given cell v its referents as encoded in v's AST. The only wrinkle when building an Excel dependence graph is that *single-cell references* (such as `A3`) and *reference ranges* (such as `A3:A10`) must be handled specially to build the dependence graph.

4.2.2 Data References

Spreadsheets typically (and Excel in particular) have two kinds of references: *single-cell references* and *range references*. A single-cell reference refers to a single

cell: for example, the formula `=A1` indicates that the formula should simply return the value stored in cell `A1`. Range references refer to multiple cells: for example, the formula `=SUM(A1:A10)` denotes that all of the values in the contiguous range of cells between `A1` and `A10` inclusive should be summed.

Note that range references are *not* limited to contiguous ranges of cells. The range `A1:A10,B1:B10` is an example of a single, valid *discontiguous* range reference combined with a comma; Excel calls this comma syntax the *range union operator*. For ease of processing, ExceLint converts all range references into a set of cell references during dependence analysis.

4.2.3 Reference Vectors

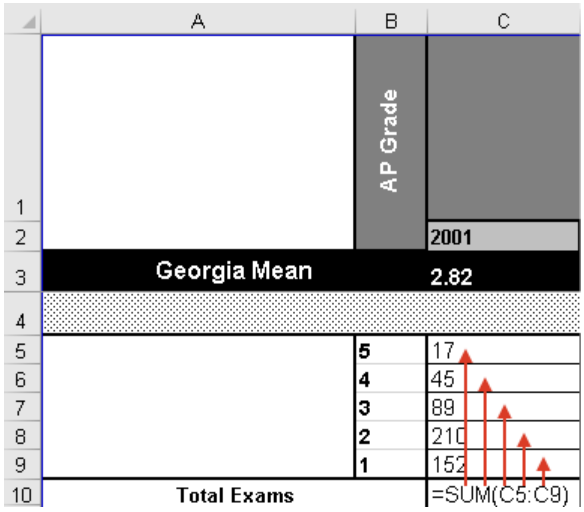


Figure 4.4: The set of reference vectors for a formula. The formula in cell `C10` "points" to data in cells `C5:C9`. Reference vectors encode this simple idea.

A *reference vector* is the basic unit of analysis in ExceLint. It encodes not just the data dependence between two cells in a spreadsheet, it also captures the spatial location of the referee and the spatial location of the referent on the spreadsheet. Reference vectors can represent all references found in Excel, even references between worksheets and other workbooks.

Intuitively, a vector can be thought of as an arrow that points from a formula to one of the formula's references. Every reference has an associated vector. For example, the formula `=SUM(C5:C9)` in cell `C10` (see Figure 4.4) "points" from cell `C10` to cells `C5` through `C9` inclusive. Vectors may cross worksheets and even workbooks (i.e., they may reference cells in other files).

4.2.3.1 Addressing Modes

Excel has two *addressing modes*, known as *relative addressing* and *absolute addressing*. Note that these are different from what Excel calls the *reference style*, which is the coordinate system used in formulas. The familiar `A1` reference style represents columns as letters and rows as integers, while the `R1C1` style represents both rows and columns as integers, starting from 1.

Relative addresses use references relative to the formula, while absolute addresses use addresses relative to the spreadsheet origin, the top left corner of the spreadsheet. Note that in Excel both address reference components, the *horizontal component* and the *vertical component*, may each have a different addressing mode. For example, the reference `$A1` has an absolute horizontal and a relative vertical component while the reference `A$1` has a relative horizontal and an absolute relative component.

Addressing modes are not useful by themselves. Instead, they are annotations that help Excel's automated copy-and-paste tools, such as Formula Fill, to generate updated addresses for copied formulas. For example, if cell $C1$ contains the formula $= \$A\$1 + B1$, and Formula Fill is employed to copy $C1$ to cells $C2$ through $C10$, the reference to $B1$ will be updated based on the location of the copy while $\$A\$1$ remains fixed. For instance, the copy in cell $C10$ will be $= \$A\$1 + B10$. Failing to use reference mode annotations correctly will cause Formula Fill to generate incorrect formula copies.

4.2.3.2 Vector Form

Reference vectors have the following form:

$$\text{ref} =< x, y, z, x', y', z', c >$$

where x, y, and z denote numerical column, row, and worksheet indices of a *referee cell* in a spreadsheet, namely, the location where the reference is used in a formula. x and y coordinates have the same semantics as Excel's alternate one-based R1C1 addressing mode. z is defined by a canonical ordering of the sheets included in an analysis, by default a lexicographical order of worksheet names, and is also one-based. x', y', and z' denote the numerical column, row, and worksheet indices of the *referent cell*, namely, the location to which the reference points. Finally, c denotes either the presence of an embedded formula constant, in which case its value is 1, or the presence of a string value, in which case its value is -1, or that the cell contains no formula, in which case its value is 0.

Every data reference in a formula has exactly one corresponding reference vector. For example, the formula `=SUM(A1:A2) + 1 + 2` in cell `B1` has four data references, `B1` $\rightarrow$ `A1`, `B1` $\rightarrow$ `A2`, `B1` $\rightarrow$ `1`, and `B1` $\rightarrow$ `2`. The reference vectors for these data references are $< 2, 1, 0, 1, 1, 0, 0 >$, $< 2, 1, 0, 1, 2, 0, 0 >$, $< 2, 1, 0, 0, 0, 0, 1 >$, and $< 2, 1, 0, 0, 0, 0, 1 >$, respectively.

Non-formula cells also have a corresponding reference vector, depending on whether their contents are *empty*, *numeric*-, or *string*-valued. For example, an empty cell in $B2$ has a single reference vector, $< 2, 2, 0, 0, 0, 0, 0 >$. A numeric cell in the same location would be $< 2, 2, 0, 0, 0, 0, 1 >$. A string-valued cell, such as a header, in the same location would be $< 2, 2, 0, 0, 0, 0, -1 >$.

4.2.3.3 Computational Agnosticism

Reference vectors are *computation agnostic*, meaning that only the spatial and structural characteristics of formulas matter. Spatio-structural analysis abstracts away the exact function calls used in a formula, for example `AVERAGE` versus `SUM`. At first glance, this appears to be a limitation. However, discerning whether a programmer intended to use `AVERAGE` instead of `SUM` requires understanding programmer intent; reporting such information runs the risk of producing more false positives. By automatically filtering out this information, spatio-structural analysis improves its precision.

An example from a real-world spreadsheet is illustrative. Cell `C114` in a financial spreadsheet[4] contains the formula `=SUM(B114+C111)`. This calculation is repeated across the row to the right. However, one cell in the calculation contains the formula `=+D114+E111`, which drops the `SUM` and introduces an extra unary plus sign. Both the use of `SUM` and the unary plus are incorrect, but neither are manifest nor latest bugs. The `SUM` adds a single number to zero, and the unary plus does nothing to the sign of the formula, so neither has any effect, nor will they in the future. While the two formulas are syntactically different, they are semantically equivalent. Because spatio-structural analysis focuses exclusively on reference behavior, it treats these formulas as identical, and reports neither of them as anomalous.

4.2.3.4 Relative Vectors

Spatio-structural analysis typically works with a *relative* form of a vector. This *relative vector* is computed differently depending on whether the horizontal or vertical component of the vector is absolute or relative, respectively. For an absolute component, the origin of the vector is the top, left corner of the spreadsheet $(1, 1)$. For a relative component, the origin is the location of the formula, (x, y, z).

---

[4] 3763250_Q304_factsheet.xls, from the EUSES corpus [42].

Consider the reference $\$D22$ for a formula located in $E3$. The horizontal component, D, which translates in R1C1 style to $x = 4$, is absolute, so the x' component of $E3$'s relative vector will be 4. The vertical component, 22, is relative, so the y' component of $E3$'s relative vector will be -18. The complete relative vector for $E3$ is $< 5, 3, 0, 4, -18, 0, 0 >$.

4.2.3.5 Reference Conversion

After the dependence analysis is run, every data dependency, including embedded constants in formulas, is converted into a relative reference vector. As with dependence analysis, the time and space complexity for this conversion is proportional to the number of references in a workbook.

4.2.4 Vector Fingerprints

ExceLint's statistical analysis requires one-vector summaries of formula reference behavior in several phases of its analysis. *Vector fingerprints* serve this purpose. A vector fingerprint is a hash function from a set of reference vectors to another vector. Spatio-structural analysis utilizes two forms of fingerprints. Both fingerprint require a computational cost proportional to the number of references being hashed together, which is usually small.

The following helper functions are useful in defining vector fingerprints.

r computes the relative vector (see "Relative Vectors" above) for a given reference vector,

$$r(< x, y, z, x', y', z', c >) =< x, y, z, x' - x, y' - y, z' - z, c >$$

t truncates the given reference vector, returning a *location-free reference vector*,

$$t(< x, y, z, x', y', z', c >) =< x', y', z', c >$$

while e zero-extends a location-free reference vector.

$$e(< x', y', z', c >) =< 0, 0, 0, x', y', z', c >$$

i makes the given reference vector *off-sheet address insensitive*.

$$i(< x, y, z, x', y', z', c >) = \begin{cases} < x, y, z, x', y', 0, c >, & \text{if } z' = 0. \\ < x, y, z, 0, 0, 1, c >, & \text{otherwise.} \end{cases}$$

z truncates the given reference vector, returning a *location-sensitive zero vector*.

$$z(< x, y, z, x', y', z', c >) =< x, y, z, 0, 0, 0, 0 >$$

4.2.4.1 Location-Free Fingerprints

The first type of fingerprint, a *location-free fingerprint*, is defined by the following hash function,

$$h_f(V) = \sum_{v \in V} t(r(i(v)))$$

where addition in this context is ordinary vector addition. Location-free fingerprints are the sum of the (off-sheet insensitive) relative components of a set of vectors.

Location-free fingerprints are chosen to have the property, when paired with Euclidean distance, such that the distance between any two formulas sharing the same location-free fingerprint is zero. In other words, two formulas in any two different locations "collide" if their location-free fingerprints are the same. This property aids in computing the frequency of a formula in a spreadsheet.

An intuitive way to think of location-free fingerprints is as *resultant vectors*, a concept borrowed from engineering and physics. Here, the relative components of a vector are added together "head to tail" to produce a single vector.

Note that the extra step of truncating z' is taken because it was observed early on that while it is useful to know that a reference is off-sheet, information about the precise location of the referent on the other sheet is not. In fact, we observed that very few off-sheet references follow any discernible pattern, even to a human eye. Omitting off-sheet location information produces a dramatic improvement on the precision of spatio-structural analysis.

4.2.4.2 Location-Sensitive Fingerprints

The second type of fingerprint, a *location-sensitive fingerprint*, is defined by the following hash function,

$$h_s(V) = z(v') + e(h_f(V))$$

where v' is any vector in V.

Location-sensitive fingerprints are chosen to have the property, when paired with Euclidean distance, such that the distance betweem two formulas sharing the same location-free fingerprint *but located in adjacent cells* is one. Euclidean distance in this context allows ExceLint to reason about formula similarity both in terms of location on a spreadsheet and reference behavior.

4.2.5 Vector Clustering

The purpose of vector clustering is to group formulas by a combination of spatio-structural and geometric factors as a basis for identifying anomalous formulas. Vector clustering utilizes location-free fingerprints for its analysis, and absent other constraints, this means that all pairs of formulas with zero distance between them belong to the same cluster. Nonetheless, an additional constraint, that all clusters must be contiguous

and rectangular, is imposed on the clustering procedure. This additional constraint is imposed because Excel's automated processing tools bias worksheets toward composing spreadsheet programs in a rectilinear fashion. Users who fight Excel in this regard find themselves with ad hoc layouts that are extraordinarily difficult to work with (see Chapter 1.1). Consequently, we rarely see non-rectilinear spreadsheet layouts outside of Excel's alternate use as a kind of "poor man's database."

Vector clustering helps answer a question about user intent, namely whether an unusual formula construct is a mistake or if it was intended. As was discussed earlier, knowing programmer intent is a virtual impossibility. Nonetheless, one way to approach the problem is to identify all potentially anomalous cells, and for each one, propose one or more fixes, and observe whether fixes improve an error model of the spreadsheet in some way.

4.2.5.1 Rectangular Decomposition

To produce a clustering of a spreadsheet amenable to our ideal error model, the clustering should have a few features. 1) Since spreadsheets containing formulas strongly encourage rectilinear composition, all clusters are rectangular. 2) Since users often utilize strings and whitespace in semantically meaningful ways, such as dividers between different computations, all cells, whether they contain strings, whitespace, numbers, or formulas should be clustered. 3) Finally, clusters should be as big as possible while still remaining meaningful, in order to capture the fact that adjacent formulas that compute the same thing are not likely to be parts of different computations. In other words, the fact that people place generally place similar formulas together is semantically meaningful, and the analysis should attempt to preserve this information.

Spatio-structural analysis employs a top-down, recursive decomposition that splits spreadsheet regions into two subdivisions according by minimizing a simple statistic,

normalized Shannon entropy [87]. Specifically, spatio-structural analysis performs a recursive binary decomposition that, at each step, chooses the split that minimizes the sum of the normalized Shannon entropy of location-free vector fingerprints in both subdivisions. Normalized Shannon entropy is employed since the binary partitioning process in no way guarantees that entropy comparisons are being made for equal-sized sets; normalization ensures that comparisons are well-behaved [13].

Normalized Shannon entropy is defined as:

$$\eta(X) = -\sum_{i=1}^{n} \frac{p(x_i) \log_b p(x_i)}{\log_b n}$$

4.2.5.2 Rectangular Decomposition Algorithm

The procedure BinaryMinEntropyTree in Figure 4.5 shows ExceLint's rectangular clustering algorithm for a given 2D worksheet, S. The procedure returns a tree; regions are stored in the leaves. A cell is an (x, y) coordinate pair. S is a set of cells, initially the entire spreadsheet. H computes the normalized Shannon entropy, where the entropy of the empty set is defined as $+\infty$. Values returns the set of distinct location-free fingerprint values for the given region. Finally, Leaf and Node are constructors for a leaf tree node and an inner tree node, respectively.

BinaryMinEntropyTree bears some resemblance to the ID3 decision tree induction algorithm from machine learning [79]. As with ID3, BinaryMinEntropyTree usually produces a good binary tree, although not necessarily the optimal (i.e., the shortest) tree. Instead, the tree is decomposed greedily, with a worst case running time proportional to the largest number of binary subdivisions of a rectangular grid.

After building the tree, clusters can be extracted by visiting the leaves of the tree and extracting the set of cells stored in each leaf. Leaves correspond to exactly one cluster.

```
BinaryMinEntropyTree(S)
 1  if l = r and t = b
 2       return Leaf(S)
 3  else
 4       Left = λx.{ cell ∈ S | fst(cell) < x}
 5       Right = λx.{ cell ∈ S | fst(cell) ≥ x}
 6       Top = λy.{ cell ∈ S | snd(cell) < y}
 7       Bottom = λy.{ cell ∈ S | snd(cell) ≥ y}
 8       EntV = λx.H(Left(x)) + H(Right(x))
 9       EntH = λy.H(Top(y)) + H(Bottom(y))
10       x = argmin_{l≤x≤r} EntV(x)
11       y = argmin_{t≤y≤b} EntH(y)
12       (p1,p2) = top, bottom
13       ent = 1.0
14       if EntV(x) ≤ EntH(y)
15            (p1,p2) = left, right
16            ent = EntV(x)
17       else
18            ent = EntH(y)
19       if ent = 0.0 and Values(p1) = Values(p2)
20            Leaf(S)
21       else
22            t1 = BinaryMinEntropyTree(p1)
23            t2 = BinaryMinEntropyTree(p2)
24            Node(t1, t2)
25  return (V, E)
```

Figure 4.5: BinaryMinEntropyTree algorithm. BinaryMinEntropyTree decomposes a spreadsheet into rectangular regions by minimizing the entropy of the distribution of vector fingerprints between splits. The procedure returns a tree; regions are stored in the leaves. A cell is an (x, y) coordinate pair. S is a set of cells, initially the entire spreadsheet. H computes the normalized Shannon entropy, where the entropy of the empty set is defined as $+\infty$. Values returns the set of distinct location-free fingerprint values for the given region. Finally, Leaf and Node are constructors for a leaf tree node and an inner tree node, respectively.

4.2.5.3 Adjacency Coalescing

While the BinaryMinEntropyTree algorithm is guaranteed to produce rectangular regions, it is not guaranteed to produce the optimal (i.e., smallest) tree. It is often the case that some adjacent cells that induce the same fingerprint are stored in different leaves of the tree. *Coalescing* merges these adjacencies subject to two rules, producing a better clustering: 1) the clusters are adjacent, and 2) the merged clusters are a contigious, rectangular region of cells.

This algorithm is a fixed-point computation, merging two regions at every step, and terminates when no more merges are possible. In the worst case, this algorithm takes time proportional to the total number of tree leaves produce by the binary decomposition algorithm. In practice, the algorithm iterates a small number of times, because the binary tree ususally produces a decomposition close to the ideal decomposition.

4.2.6 Proposed Fixes

When a user fixes a reference bug in a formula, that formula's fingerprint, which summarizes its reference behavior, changes. If the user makes a change such that the corrected formula now maintains a reference invariant, this change will be reflected in the fingerprint clustering when the user re-runs the spatio-structural analysis. The changed formula will have "joined" or "merged" with the cluster of other cells that maintain the same reference invariant.

The purpose of *proposed fixes* is to explore the effect of such fixes. A *proposed fix* is an operation that mimics the effect of "correcting" a formula. Since the working hypothesis of the analysis is that unusual formulas are wrong, and that other, more common formulas are likely correct, spatio-structural analysis leverages this fact to identify which cells should be fixed, and how. Those fixes that do not cause formulas to join existing invariant clusters are not likely to be good fixes.

Because running a new spatio-structural analysis for every proposed fix—of which there may be hundreds or thousands for a given worksheet—is an expensive operation, the analysis simulates the effect of the change instead. Since we know what the effect will be, namely that the fixed cell joins an existing invariant cluster, the analysis simply "moves" cells from one cluster to another. Exploring fixes in this manner is relatively inexpensive because most of the existing analysis can be reused.

Formally, a proposed fix is the tuple (s, t), where s is a (nonempty) set of *source cells* and t is a (nonempty) set of *target cells*. t must always be an existing cluster but s may not be; source cells may be borrowed from other clusters. This pair should be thought of as as operation that *replaces the fingerprints* of cells in s with those from cells in t. Replacing fingerprints simulates the effect of a user changing the formula: when a formula changes, so does its fingerprint.

Not all proposed fixes are good, and some are likely bad. Spatio-structural analysis uses an error model to identify which fixes are the most promising. The error model is discussed in the next section.

4.2.7 Entropy-Based Error Model

The purpose of an error model is to explore the potential effect of correcting reference errors in a spreadsheet. The model should help both identify errors and to see the impacts of correcting them.

Spatio-structural analysis uses an entropy-based model. The intuition of the model is that reference errors, which result in irregularities in the rectangular clustering, increase entropy *relative to the same spreadsheet without errors*. A proposed fix that reduces entropy may be a good fix because it moves the erroneous spreadsheet closer to the correct spreadsheet.

Since most formulas belong to large rectangular clusters, those that do not are unusual and are likely good candidate fixes. The model allows the analysis to explore

the impact of fixing these irregularies—making the spreadsheet more rectangular—choosing only the most promising ones which are then presented to the user.

Formally, a model m is a set of cells in a worksheet (i.e., a rectangular clustering). A set of proposed fixes of size n are a set of new models $m'_1 \ldots m'_n$, where each m'_i contains one proposed fix $(s,t)_i$. The *impact* of fix the $(s,t)_i$ is defined as the difference in normalized Shannon entropy,

$$\delta\eta_i = \eta(m_i) - \eta(m)$$

Positive values of $\delta\eta_i$ correspond to increases in entropy and suggest that a proposed fix is bad. Negative values of $\delta\eta_i$ correspond to decreases in entropy and suggest that a proposed fix is good.

Somewhat counterintuitively, fixes that result in large decreases in entropy are worse than fixes that result in small decreases. A fix that changes large swaths of a spreadsheet will result in a large decrease in entropy, but this is not a good fix for several reasons. First, the central hypothesis of anomaly analysis is that rare anomalies are most indicative of errors. Since rare anomalies by definition make up only a small proportion of a spreadsheet, fixing them should result in small (but non-zero) decreases in entropy. The best fixes are those where the prevailing pattern is a strong signal and so that corrections are minor. Second, large fixes are more work. A primary goal is spatio-structural analysis is to maximize the user effort, steering users toward those likely errors that are the hard to find and that maximize effort.

Informally, reference vector fingerprint clusters can be thought of as "summaries" of the reference behavior of a spreadsheet. We assume that the programmer understands the task at hand, and therefore, in general, the spreadsheet is well structured. Nonetheless, we know from empirical studies of spreadsheet errors (see Chapter 4.1) that users are likely to make a small number of errors, proportional to the size of the spreadsheet. We seek those fixes that preserve the overall structure of the spreadsheet.

An analogy to data compression is helps understand the intuition. Lossy data compression, which also often utilizes the same entropy-based insight, seeks a similar goal. The JPEG compression standard tries to find large regions of similar pixels, which are then summarized, saving space, while preserving the overall appearance of the image. Information loss is tolerable as long as the overall appearance of the image is preserved.

4.2.7.1 Re-clustering

Executing a proposed fix yields a new model $\tilde{m}_i$. This model is not yet the desired outcome, as it is possible that a proposed fix causes adjacent clusters to change. For example, a single error in the middle of a column of similar formulas will yield three regions when clustered. A fix that adds such an anomaly into a cluster should split the cluster, whereas a fix that does the converse should merge the neighboring clusters. As a result, clustering and coalesing steps need to be repeated after proposed fixes are executed, yielding the new model m_i.

This is potentially an expensive operation. Fortunately, the binary tree decomposition can be used sparingly under certain conditions. A *split* is defined as a fix that produces an additional number of clusters after a fix operation. A fix that removes a cell from the end of a cluster and moves it into an adjacent cluster is not a split because the number of clusters does not increase. A fix that splits a cluster cannot reduce entropy because more clusters cannot have lower entropy than fewer clusters. Furthermore, the split is limited to the source cluster; merging a cell into a target will never cause a split in the target. The binary tree decomposition algorithm only needs to be run on source clusters to find new splits when a split would occur. Any admissible (i.e., rectangular) merges resulting from a fix will be found by the coalescing fixpoint algorithm, which is always run after a fix.

4.2.7.2 Producing a Set of Fixes

Spatio-structural analysis considers all possible fixes for every reference invariant cluster t–those cells that share a fingerprint–in the spreadsheet. Every cluster t is paired with adjacent s, and the fix (s, t) is considered to be a candidate subject to four conditions (below).

The analysis often produces far more proposed fixes than either the user is likely to want to see. In many cases, there are also far more fixes than the likely number of bugs in the spreadsheet. Some fixes are independent; applying fix f_i and f_j in sequence has the same effect as applying them in the reverse order. Other fixes are not independent; for instance, the analysis sometimes proposes more than one fix utilizing the same source. Clearly, it is not possible to perform both fixes. As a result, the analysis suppresses certain fixes, scores fixes by a fitness function, and then ranks the fixes in order from most to least promising (i.e., in descending order by score), thresholds the ranking and returns all proposed fixes below the threshold.

The cutoff threshold is a user-defined parameter that represents the proportion of the worksheet that a programmer is willing to inspect. The default value, 5%, is based on the observed frequency of spreadsheet errors in the wild (see Chapter 4.1). The programmer may adjust the threshold to inspect more or fewer cells, depending on their preference. In all cases, spatio-structural analysis returns the most promising fixes below the cutoff.

4.2.7.2.1 Condition 1: Rectangularity. The first condition is that fixes produce rectangular layouts. This condition arises from the fact that Excel and other spreadsheet languages have many affordances for rectangular composition of functions. For instance, many functions—like `SUM`—are designed to take a single input vector. An input vector in Excel is usually specified by the user with "range" syntax ":", such as `A1:A10`, which denotes a contiguous, rectangular region that includes all cells between cell `A1` and cell `A10`, inclusive. While it is possible in many cases to

supply discontiguous regions using Excel's "reference union" operator "`,`", such as `A1,A5,A7:A10`, doing so is cumbersome and is not supported by all functions that take ranges. Spatio-structural analysis biases the search toward fixes (s, t) such that $s \cup t$ is rectangular.

4.2.7.2.2 Condition 2: Compatible Datatypes. Likely anomalies are those identified by fixes m_i that produce small, positive values of $\delta\eta_i$. Nonetheless, this is not a sufficient condition to identify an anomaly. Small clusters can belong to data of any type (string data, numeric data, whitespace, and other formulas). Fixes between clusters of certain datatypes are not likely to produce desirable effects. For instance, while a string may be replaced with whitespace and vice-versa, neither of these proposed fixes have any effect on the computation, and are not formula-related errors. Furthermore, while proposed fixes from strings and whitespace do indeed identify certain kinds of bugs—typos and missing formulas—both of these fixes produce large numbers of false positives because formulas are frequently adjacent to large whitespace clusters (spreadsheet edges) and string clusters (headers).

We consider only proposed fixes of the following types: 1) fixes from formulas to formulas, 2) fixes from numbers to formulas, and fixes from 3) formulas to numbers. The first category includes classic formula data reference errors, such as omitting a reference or a constant. The second category includes cases where a formula was mistakenly omitted or where the computation is suspicious (i.e., "fudging" numbers so that they add up to 100). The third category includes cases where formulas are unexpectedly inserted into a primarily numeric region of a spreadsheet, which may indicate either a copy-and-paste error, a suspicious computation (i.e., a converse method of "fudging" numbers), or where the entire region itself probably should be computed (i.e., a "bad neighborhood" of values).

4.2.7.2.3 Condition 3: Inherent Computation Anomalies. Two common forms of computation are inherently unusual in the spatio-structural domain: computation chains, and aggregate computations.

A *computation chain* is a data cell d and an ordered sequence of formulas $f_0, \ldots, f_n$ such that f_0 refers to d and each $f_i, i > 0$ refers to f_{i-1}. This form of constructive computation is commonly used in functional programs. For instance, the `fold` operation seen in many functional progrmming languages is a computation chain. Since spreadsheet languages like Excel are also functional, it is not surprising that we see such structures arise in spreadsheets.

During spatio-structural analysis, d almost always ranks highly in anomalousness. It often sits adjacent to a large cluster of formulas, namely the computation chain. For example, spreadsheet users often number the rows of a spreadsheet with the first row starting at 1 and with subsequent rows refering to the previous one, as in `=A1+1`, `=A2+1`, ..., `A`N `+ 1`. Since d is usually a false positive, spatio-structural analysis excludes any proposed fix with $\{d\}$ as its source and $\{f_0, \ldots, f_n\}$ as its target.

An *aggregate computation* is a formula cell f_a and a set of input cells $c_0, \ldots, c_n$ such that f_a refers exclusively to $c_0, \ldots, c_n$. Again, this is a form of computation commonly seen in functional languages, particularly in languages with a statistical flavor like R. Excel comes with a large set of built-in statistical functions, so functions of this sort are invoked frequently. Examples built-in functions are `SUM`, `AVERAGE`, and so on, and so this is also a common construction seen in spreadsheets.

During spatio-structural analysis, f_a also frequently ranks highly in anomalousness. Like with computation chains, f_a often sits adjacent to $c_0, \ldots, c_n$, and so f_a appears unusual. Since f_a is usually a false positive, spatio-structural analysis excludes any proposed fix with $\{f_a\}$ as its source and $\{c_0, \ldots, c_n\}$ as its target.

Note that eliminating these false positives does not hurt the ability of the analysis to find many bugs. For example, off-by-one references are extremely common in

spreadsheets, where an aggregation like `SUM` refers to either *one more* or *one fewer* element. Formally, f_a refers to $c_0, \ldots, c_n, d$ (one more) or $c_0, \ldots, c_{n-1}$ (one fewer). Since spatio-structural analysis only excludes aggregates that refer *exclusively* to $c_0, \ldots, c_n$, it still catches these common errors.

4.2.7.2.4 Condition 4: Unlikely Fixes. While high-entropy formula layouts are often indicative of bugs, programmers sometimes do produce error-free high-entropy layouts. Layouts that fall into this category can be thought of as "bucking the trend" to produce rectangular computations. Spreadsheets produced in this way will repeatedly break the inferred reference invariants produced by the clustering stage of the analysis. Consequently, without corrective action, the analysis will propose many unnecessary fixes.

How do we detect such a situation? Since the central hypothesis of anomaly detection is that only unusual errors should be reported, one way is to ensure that all proposed fixes fix only rare problems. When a proposed fix appears frequently, it is more likely to be indicative of an unusual layout than a bug.

For example, a common spreadsheet pattern is to use the rightmost column of a table for summary purposes. Each row sums the cells to the left, and the rightmost, bottommost cell is a sum-of-sums of the cells above it. ExceLint expects the rightmost, bottommost sum to be anomalous (see "Inherent Computation Anomalies" below). However, if a programmer *repeatedly places* the sum-of-sums elsewhere, the analysis may not recognize the construction as inherently anomalous and may suggest fixing it even though it is common in the spreadsheet. A real-world example of such a case is shown in Figure 4.6. The pattern shown was widespread in this particular spreadsheet, occuring more than 10 times. The *regularity map*, a kind of visualization of the spatial clustering produced by the analysis (see Chapter 4.3.1.1) is shown in Figure 4.7.

	B	C	D	H	I	J
37	**Technology**					=H37/$H$113
38		Computer Equipment	4500	4500		=H38/$H$113
39		Other tech equip/services	0	2000		=H39/$H$113
40		**Total Technology Exp**	**=SUM(D38:D39)**		**=SUM(H38:H39)**	**=I40/$H$113**
41						
42	**Music Programs**					
43		Instrumental Music (4th/5th)	100	7100		=H43/$H$113
44		Music/Choir	500	500		=H44/$H$113
45		**Total Music Exp**	**=SUM(D43:D44)**		**=SUM(H43:H44)**	**=I45/$H$113**
46						
47	**Library**					
48		Library Materials	4000	4000		=H48/$H$113
49		Writers Celebration	200	200		=H49/$H$113
50		Speech Contest (4th/5th)	150	150		=H50/$H$113
51		**Total Library Exp**	**=D48**		**=SUM(H48:H50)**	**=I51/$H$113**
52						
53	**Science and Math**					
54		Science Lab coord & supplies	7400	=7875+250		=H54/$H$113
55		Science Fair coord & supplies	4200	=4200+125		=H55/$H$113
56		Math Olympiad	575	575		=H56/$H$113
57		Science/Env. Projects	0	1000		=H57/$H$113
58		**Total Science & Math Exp**	**=SUM(D54:D57)**		**=SUM(H54:H57)**	**=I58/$H$113**

Figure 4.6: A spreadsheet with an unusual invariant. The spreadsheet shown is from the EUSES corpus [42]. The `SUM` formulas in column `I` are all off-center, computing the sum of the values to the left and above in column `H`. These formulas compute the correct value, but the spatio-structural invariant is difficult to infer because it is not rectilinear.

Spatio-structural analysis corrects for overrepresented fixes by computing the probability of a fix when drawn uniformly from all possible fixes for a spreadsheet. If a given set of highly-ranked fixes is unlikely given their probabilty of being drawn at random, then we conclude that there is bias toward a particular fix. Bias suggests that the underlying problem is a repeated unusual pattern and not a bug. Those fixes with strong biases are rejected.

Formally, let F be a set of fixes of size m. Recall that a fix is the tuple (s, t) where s is a set of cells that violate a spatio-structural invariant and t is a set of cells that do not. (s, t) represents the operation to correct each formula in $s_i \in s$ such that $h_f(s_i) = h_f(t_j)$ for any formula $t_j \in t$ (since $\forall t_i, t_j \in t, h_f(t_i) = h_f(t_j)$). The probability of seeing a given fix is the multinomial probability $p_1, \ldots, p_k$ where k

	B	C	D	H	I	J
37	**Technology**					=H37/$H$113
38		Computer Equipment	4500	4500		=H38/$H$113
39		Other tech equip/services	0	2000		=H39/$H$113
40		**Total Technology Exp**	**=SUM(D38:D39)**		**=SUM(H38:H39)**	**=I40/$H$113**
41						
42	**Music Programs**					
43		Instrumental Music (4th/5th)	100	7100		=H43/$H$113
44		Music/Choir	500	500		=H44/$H$113
45		**Total Music Exp**	**=SUM(D43:D44)**		**=SUM(H43:H44)**	**=I45/$H$113**
46						
47	**Library**					
48		Library Materials	4000	4000		=H48/$H$113
49		Writers Celebration	200	200		=H49/$H$113
50		Speech Contest (4th/5th)	150	150		=H50/$H$113
51		**Total Library Exp**	**=D48**		**=SUM(H48:H50)**	**=I51/$H$113**
52						
53	**Science and Math**					
54		Science Lab coord & supplies	7400	=7875+250		=H54/$H$113
55		Science Fair coord & supplies	4200	=4200+125		=H55/$H$113
56		Math Olympiad	575	575		=H56/$H$113
57		Science/Env. Projects	0	1000		=H57/$H$113
58		**Total Science & Math Exp**	**=SUM(D54:D57)**		**=SUM(H54:H57)**	**=I58/$H$113**

Figure 4.7: An example with high entropy invariant clusters. The same spreadsheet as shown in Figure4.6 using the *regularity map* visualization (see Chapter 4.3.1.1). `I51` and `I58` sum over differently sized ranges than `I40` and `45` so their spatio-structural invariants are different and so are their colorings on the regularity map. Because they also have an unusual spatial relationship with their references, the entropy model will propose them as potential fixes. Since the pattern is common on *this* spreadsheet, ExceLint supresses reporting them.

is the number of distinct fingerprints there are in a spreadsheet, and each p_i is the likelihood of of seeing a fix for a given fingerprint. Let c be the maximum number of fixes generated for a fingerprint cluster; this number is a parameter of the formula fix search algorithm. We set $c = 4$ in the current algorithm, since there are only four ways to expand a rectangular cluster by merging with an adjacent rectangle such that merge remains rectangular[5]. If there are m_i fingerprint clusters for a given fingerprint, then there are $c \cdot m_i$ possible fixes for that fingerprint. The probability of selecting a fix for

[5]Other variations of the algorithm might search "further", by considering transitive adjacencies, or by abandoning the constraint that merges must remain rectangular.

fingerprint i at random is $\frac{cm_i}{\sum_{j=1}^{k} cm_j} = \frac{m_i}{m}$ where k is the number of fingerprint clusters. The number of trials for the multinomial is n, the maximum number of potential fixes returned to the user (i.e., the user-defined cutoff). By default, $n = 0.05 \cdot |C|$, where C is the number of cells in the spreadsheet.

Let $X = (X_1 = x_1, \ldots, X_k = x_k)$ be the observed fingerprint counts for a sample of n proposed fixes. To determine whether fingerprint i is overrepresented—whether the count x_i is extreme—compute the probability $t = P(X_i \geq x_i)$. If $t \leq \alpha$ then reject the null hypothesis and suppress fixes with fingerprint i. By default, ExceLint uses $\alpha = 0.05$, which corresponds to a 95% confidence level.

4.2.7.3 Fix Distance

Among fixes with an equivalent entropy reduction, some fixes are better than others. For instance, when copying and pasting formulas, failing to update one reference is more likely than failing to update all of them, since the latter has a more noticable effect on the computation. Therefore, a desirable criteria is one that favors smaller fixes in location-sensitive vector fingerprint space.

The following distance is inspired by the earth mover's distance [71].

$$d(x, y) = \sum_{i=1}^{n} \sqrt{\sum_{j=1}^{k} (h_s(x_i)_j - h_s(y_i)_j)^2}$$

where x and y are two models, where n is the number of cells in both x and y, where h_s is a location-sensitive fingerprint hash function, where i indexes over the same cells in both x and y, and where j indexes over the vector components of a fingerprint vector for fingerprints of length k. As a shorthand, we denote d_i to refer $d(m, m_i)$.

4.2.7.4 Entropy Reduction Impact Score

The quality of a fix is determined by an *entropy reduction impact score*, S_i, which computes the potential improvement between the original model, m, and the fixed model, m_i,

$$S_i = \frac{n_t}{-\delta\eta_i d_i}$$

where n_t is the size of the target cluster, $\delta\eta_i$ is difference in entropy from m_i to m, and d is the fix distance.

The best fixes are found by maximizing S_i. Since the best fixes minimize $-\delta\eta_i$, such fixes maximize S_i. Likewise, "closer" fixes according to the distance metric also produce higher values of S_i. Finally, the score states a preference for fixes whose "target" is a large cluster, what we call *invariant significance*. This preference ensures, in keeping with the central hypothesis of anomaly analysis, that the highest ranked deviations are actually rare with respect to a mined invariant.

4.2.7.5 Ranking and Thresholding

After the algorithm produces a set of candidate fixes, fixes that do not meet the four conditions above are suppressed. The remaining set of fixes are ranked by their entropy reduction impact score. Finally, the set of ordered fixes is thresholded by the user parameter and returned to the user.

4.3 ExceLint Implementation

ExceLint is written in a mixture of C# and F# for the .NET managed language runtime, written as a plugin for Microsoft Excel using the Visual Studio Tools for Office (VSTO) interoperability framework. It is compatible with Excel versions 2010-2016.

We used the same parser[6] and dependence analysis[7] libraries we developed for CheckCell in the development of ExceLint. Both libraries were extended to support additional features needed by ExceLint[8].

4.3.1 Visualizations

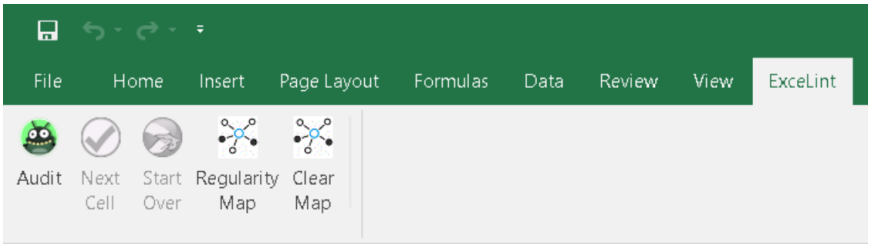


Figure 4.8: The ExceLint toolbar. The "Regularity Map" button displays the regularity map. The "Audit" button performs a cell-by-cell audit. See Sections 4.3.1.1 and 4.3.1.2.

ExceLint has two visual tools that assist users to find bugs. Both tools are based on a spatio-structural analysis of the spreadsheet. The first is the *regularity map* tool, which is described in Chapter 4.3.1.1. The other is the *proposed fix* tool described in Chapter 4.3.1.2.

4.3.1.1 Regularity Map

The *regularity map* is a visualization for finding potential bugs in spreadsheets and is the primary contribution of this work. The map takes advantage the keen human ability to quickly spot deviations in visual patterns. A sample spreadsheet is shown in Figure 4.9 and its corresponding regularity map is shown in Figure 4.10.

---

[6]https://github.com/plasma-umass/parcel

[7]https://github.com/dbarowy/Depends

[8]For example, CheckCell did not need to support the peculiarities of Excel address mode annotations, which ExceLint does.

	A	B	C	D	E	F	G	H	I
1									Christa Posey
2		ACME Toy Employee Payroll							36182
3									
4		Hours worked in December							
5		*Week 1*	*Week 2*	*Week3*	*Week 4*	*Total Hours*	*Overtime Hrs*		
6	**Green**	10	10.5	5.25	8.58	=SUM(B6:E6)	=MAX(B6-40,0)		
7	**Smith**	15	18	20.5	=AVERAGE(B7:D7)	=SUM(B7:D7)	=MAX(B7-40,0)		
8	**Jones**	13	21.5	16	=AVERAGE(B8:D8)	=SUM(B8:D8)	=MAX(B8-40,0)		
9	**Adams**	42.5	38	43	=AVERAGE(B9:D9)	=SUM(B9:D9)	=MAX(B9-40,0)+MAX(C9-40,0)+MAX(D9-40,0)		
10	**Stevens**	44	40	48	=AVERAGE(B10:D10)	=SUM(B10:D10)	=MAX(B10-40,0)+MAX(C10-40,0)+MAX(D10-40,0)		
11	**Harris**	23	45	38.5	=AVERAGE(B11:D11)	=SUM(B11:D11)	=MAX(B11-40,0)+MAX(C11-40,0)+MAX(D11-40,0)		
12									
13	**Weekly Totals**	=SUM(B6:B11)	=SUM(C6:C11)	=SUM(D6:D11)		=SUM(F6:F11)			
14	**Max Hours**	44	45	48					
15									
16		Hourly Rate	Gross Pay	Taxes	Net Pay				
17	**Green**	7.5	=(F6+G6)*B17	28.96	=(C17-D17)				
18	**Smith**	7.25	=(F7+G7)*B18	=(C18*0.15)	=(C18-D18)				
19	**Jones**	8.5	=(F8+G8)*B19	=(C19*0.15)	=(C19-D19)				
20	**Adams**	8.25	=(F9+G9)*B20	*f*	=(C20-D20)				
21	**Stevens**	10.5	=(F10+G10)*B21	=(C21*0.15)	=(C21-D21)				
22	**Harris**	9.5	=(F11+G11)*B22	=(C22*0.15)	=(C22-D22)				
23									
24	**Totals**		=SUM(C17:C22)	=(C24*0.15)	=(C24-D24)				

Figure 4.9: A buggy spreadsheet shown with Excel's formula view. This spreadsheet was drawn from the FUSE corpus [9]. This sheet contains a number of irregularities, many of which are not obvious even in formula view.

	A	B	C	D	E	F	G	H	I
1									Christa Posey
2		ACME Toy Employee Payroll							23-Jan-03
3									
4		Hours worked in December							
5		*Week 1*	*Week 2*	*Week3*	*Week 4*	*Total Hours*	*Overtime Hrs*		
6	**Green**	10.00	10.50	5.25	8.58	34.33	0.00		
7	**Smith**	15.00	18.00	20.50	17.83	53.50	0.00		
8	**Jones**	13.00	21.50	16.00	16.83	50.50	0.00		
9	**Adams**	42.50	38.00	43.00	41.17	123.50	5.50		
10	**Stevens**	44.00	40.00	48.00	44.00	132.00	12.00		
11	**Harris**	23.00	45.00	38.50	35.50	106.50	5.00		
12									
13	**Weekly Totals**	147.50	173.00	171.25		500.33			
14	**Max Hours**	44.00	45.00	48.00					
15									
16		Hourly Rate	Gross Pay	Taxes	Net Pay				
17	**Green**	$7.50	$257.48	$28.96	$228.52				
18	**Smith**	$7.25	$387.88	$58.18	$329.69				
19	**Jones**	$8.50	$429.25	$64.39	$364.86				
20	**Adams**	$8.25	$1,064.25	*f*	#VALUE!				
21	**Stevens**	$10.50	$1,512.00	$226.80	$1,285.20				
22	**Harris**	$9.50	$1,059.25	$158.89	$900.36				
23									
24	**Totals**		$4,710.10	706.515	$4,003.59				

Figure 4.10: A buggy spreadsheet shown using ExceLint's regularity map. This is the same spreadsheet shown in Figure 4.9. Irregularities become apparent because the map's brightly-colored *invariant blocks* draw attention to unevenly-sized clusters. Cells `E6`, `F6`, `D17`, and `D20` are suspicious and warrant further investigation. In fact, all of these visual cues are indicative of real problems.

The purpose of the regularity map is to draw attention to irregularities in the spreadsheet. Each colored block, an *invariant block*, represents a contiguous region where a single formula reference behavior is observed. While it can sometimes be easy to see that a particular invariant is maintained or not in formula view, such as the

discontiguity between cell `E6` and cells `E7:E11`, in other cases, these differences are subtle, as in the discontiguity between cell cell `F6` and cells `F7:F11`. In fact, cell `F6` sums over one additional cell.

While the underlying clustering is strictly rectangular for the purposes of entropy modeling, we made the decision to reuse the same color in the visualization anywhere the same vector fingerprint is used. For example, all the numeric data in the visualization are shown using the same shade of blue, representing the fact that all the cells in that region contain the same kind of data. Whitespace and string clusters are also omitted from the visualization as they mostly distract the eye.

Colors are chosen to maximize perceptual differences. The runtime attempts to assign colors such that adjacent clusters use a complementary or near-complementary colors. This is essentially a graph coloring problem. The algorithm works by building a graph of all adjacent clusters, then by coloring them using a greedy coloring heuristic, largest degree ordering [91]. This scheme does not produce an optimal coloring, but it does have the benefit of running in $O(n)$ time, where n is the number of vertices in the graph.

New colors are chosen according to the following scheme. Colors are represented internally using the Hue-Saturation-Luminosity (HSL) model, which models all representable colors on a computer as a cylinder (see Figure 4.11). Starting from a point on the circle at the end of the cylinder, hue is the angle around this circle. Saturation is a number from 0 to 1 and represents the distance from the center point of the circle, with 0 being at the center. Luminosity is also a number between 0 and 1 and represents a point along the length of the cylinder.

The algorithm begins by choosing colors at the starting point of hue = 180°, saturation 1.0, and luminosity 0.5. This corresponds to bright blue. Subsequent colors are chosen with saturation and luminosity fixed, but with the hue being the value that maximizes the distance on the hue circle between the previous color and any

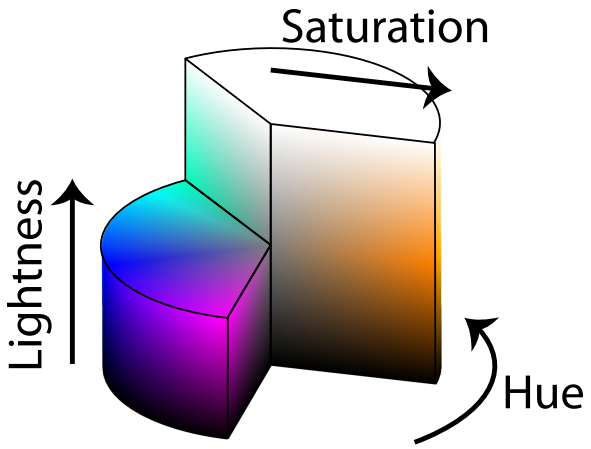


Figure 4.11: The Hue-Saturation-Luminosity (HSL) color model. Color is represented as a point in a cylindrical space, with hue corresponding to the angle around the circle, saturation to the length along the radius, and luminosity to the length along the cynlinder. Image ©2010 Jacob Rus (CC BY-SA 3.0). The original image was cropped.

other color. For example, the next color would be HSL(180°, 1.0, 0.5) followed by HSL(0°, 1.0, 0.5) and then HSL(90°, 1.0, 0.5) (see Figure 4.12). The algorithm is also parameterized by a color restriction so that other colors may be used in overlays. For instance, our algorithm currently omits bright red (HSL(0°, 1.0, 0.5)) as we plan to add additional imformation in the future using that color.

4.3.1.2 Proposed Fixes

Another visualization, the *proposed fix tool*, automates some of the human intuition that makes the regularity map an effective tool. This visualization is essentially a cell-by-cell audit of the highest-ranked proposed fixes generated in the third phase of the spatio-reference analysis described in Chapter 4.2.7. Figure 4.13 shows an example proposed fix. The portion in red represents a potential error, and the portion in green represents the set of formulas that ExceLint thinks correctly maintains the invariant. This fix is in fact a good suggestion, as the cell in `D17` is missing a formula.

Figure 4.14 shows the complete set of proposed fixes generated for the same spreadsheet as the one shown in Figures 4.9 and 4.10. Note that this visualization

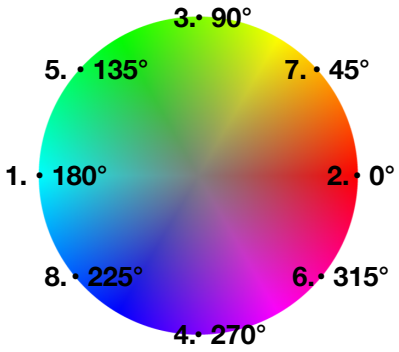
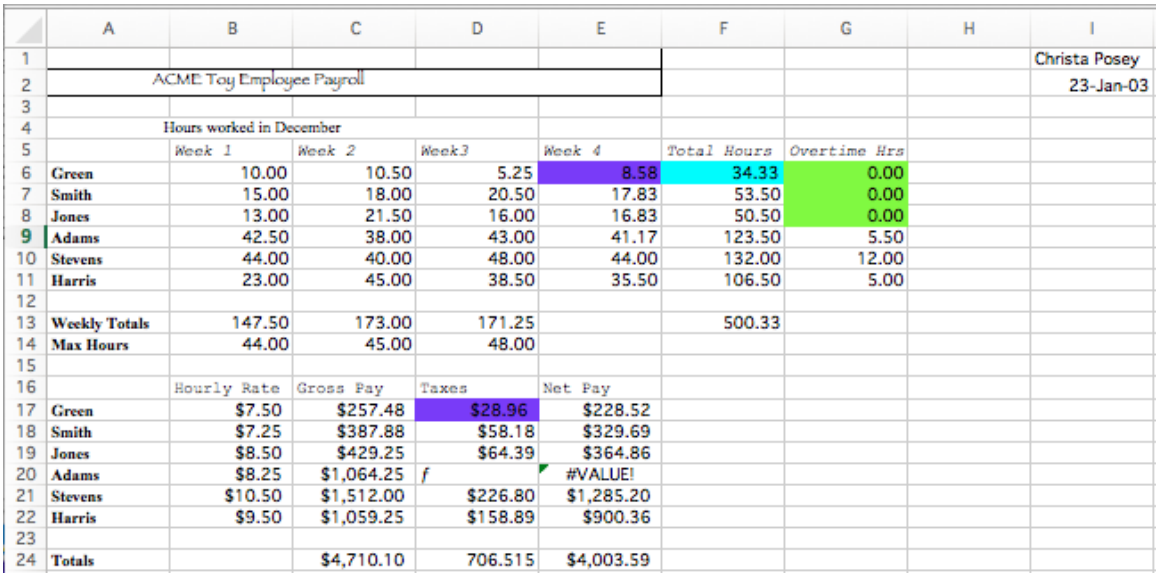


Figure 4.12: ExceLint uses complementary colors to highlight adjacent clusters. The order of colors chosen is shown in the image. Each adjacent color is guaranteed to the have the furthest angular distance from all previously chosen colors. Derived from image ©2010 Jacob Rus (CC BY-SA 3.0).

	A	B	C	D	E	F	G	H	I
1									Christa Posey
2		ACME Toy Employee Payroll							23-Jan-03
3									
4		Hours worked in December							
5		*Week 1*	*Week 2*	*Week3*	*Week 4*	*Total Hours*	*Overtime Hrs*		
6	**Green**	10.00	10.50	5.25	8.58	34.33	0.00		
7	**Smith**	15.00	18.00	20.50	17.83	53.50	0.00		
8	**Jones**	13.00	21.50	16.00	16.83	50.50	0.00		
9	**Adams**	42.50	38.00	43.00	41.17	123.50	5.50		
10	**Stevens**	44.00	40.00	48.00	44.00	132.00	12.00		
11	**Harris**	23.00	45.00	38.50	35.50	106.50	5.00		
12									
13	**Weekly Totals**	147.50	173.00	171.25		500.33			
14	**Max Hours**	44.00	45.00	48.00					
15									
16		Hourly Rate	Gross Pay	Taxes	Net Pay				
17	**Green**	$7.50	$257.48	$28.96	$228.52				
18	**Smith**	$7.25	$387.88	$58.18	$329.69				
19	**Jones**	$8.50	$429.25	$64.39	$364.86				
20	**Adams**	$8.25	$1,064.25	*f*	#VALUE!				
21	**Stevens**	$10.50	$1,512.00	$226.80	$1,285.20				
22	**Harris**	$9.50	$1,059.25	$158.89	$900.36				
23									
24	**Totals**		$4,710.10	706.515	$4,003.59				

Figure 4.13: The proposed fix tool in use. A flagged bug is labeled in red, with neighboring cells that maintain a different reference invariant labeled in green.

is only shown here for the purposes of understanding the entire set of admissible proposed fixes. ExceLint does not show all of these fixes to the user during the audit. The cutoff threshold for this spreadsheet shows only the first two highest-ranked outliers, `D17` and `F9`. Both are real bugs.



	A	B	C	D	E	F	G	H	I
1									Christa Posey
2		ACME Toy Employee Payroll							23-Jan-03
3									
4		Hours worked in December							
5		Week 1	Week 2	Week3	Week 4	Total Hours	Overtime Hrs		
6	Green	10.00	10.50	5.25	8.58	34.33	0.00		
7	Smith	15.00	18.00	20.50	17.83	53.50	0.00		
8	Jones	13.00	21.50	16.00	16.83	50.50	0.00		
9	Adams	42.50	38.00	43.00	41.17	123.50	5.50		
10	Stevens	44.00	40.00	48.00	44.00	132.00	12.00		
11	Harris	23.00	45.00	38.50	35.50	106.50	5.00		
12									
13	Weekly Totals	147.50	173.00	171.25		500.33			
14	Max Hours	44.00	45.00	48.00					
15									
16		Hourly Rate	Gross Pay	Taxes	Net Pay				
17	Green	$7.50	$257.48	$28.96	$228.52				
18	Smith	$7.25	$387.88	$58.18	$329.69				
19	Jones	$8.50	$429.25	$64.39	$364.86				
20	Adams	$8.25	$1,064.25	f	#VALUE!				
21	Stevens	$10.50	$1,512.00	$226.80	$1,285.20				
22	Harris	$9.50	$1,059.25	$158.89	$900.36				
23									
24	Totals		$4,710.10	706.515	$4,003.59				

Figure 4.14: The complete set of candidate bugs for an example spreadsheet. This visualization is for discussion purposes and is not shown to the user. In this image, all of highlighted cells indicate true reference bugs. With the threshold to show only the most unusual 5%, only two of the highlighted bugs are shown.

While the regularity map visualization is clearly more informative—even with the proposed fix tool, it is often useful to see the regularity map as context when deciding why a fix should be made—the proposed fix tool is best suited for large spreadsheets. When a user clicks the Audit button using the regularity map, they must visually inspect all sheets. If a spreadsheet is large or has many sheets, this may be an onerous task. The proposed fix tool instead prioritizes likely reference bugs, highlighting and centering the user's view on each one, one at a time. If a sheet has no likely bugs, unlike the regularity map, the proposed fix tool shows nothing.

4.3.2 Optimizations

Obtaining a level of performance sufficient for interactivity was a challenge during ExceLint's development. This section describes some of the performance optimizations undertaken to make ExceLint fast enough to meet this important usability goal.

4.3.2.1 Programming Language Issues

ExceLint is written in two .NET managed languages, C# and F#, and interoperate with Excel using the Visual Studio Tools for Office (VSTO) framework. Excel is written in a native language programming language (C++ [69]) and utilizes Microsoft's Component Object Model (COM) native language framework to represent objects [68]. While VSTO and COM greatly simplify sharing data between .NET and Excel, as COM objects have direct analogues in .NET, these frameworks introduce a number of performance pitfalls.

Because ExceLint and Excel use different programming language runtimes (.NET vs COM), they necessarily reside in different operating system processes. While VSTO enables almost[9] transparent use of COM objects in .NET code, it provides access to these objects by marshaling them into .NET; in other words, read objects are always copied. COM object read calls between ExceLint and Excel often take on the order of tens of milliseconds per call. This fact supplies a strong incentive to minimize the volume of object copying between Excel and ExceLint.

ExceLint avoids such overheads by eagerly marshaling all objects that it may need to read using bulk-read functionality available in VSTO. Communication between ExceLint and Excel occurs at the beginning of an analysis and then at the end when results need to be returned. Sadly, fine-grained progress bars, which are a nice affordance for users, are difficult to implement in a high-performance manner. All user interface updates in Excel must be run "in-process", namely in the single COM thread in which Excel resides, which requires an expensive cross-domain call.

[9]Because COM and .NET use different garbage collection algorithms, in practice programmers need to be *keenly* aware of the differences. COM uses a reference counting collector whereas .NET uses a generational mark-sweep collector. When .NET holds a reference to a COM object, COM also maintains a reference. Unfortunately, Excel refuses to shut down until the reference count on all objects shared with .NET is zero. Since .NET provides no guarantees about *when* an unreferenced object is to be collected, programmers (including myself) must resort to a number of hacks to encourage the runtime to collect these objects sooner rather than later, such as nulling their fields and manually calling `System.GC.Collect`.

ExceLint skirts this performance trap by communicating as few progress points back to the UI as possible, essentially one at the end of each analysis phase. The progress bar is correspondingly coarse. These problems could have been mitigated by writing ExceLint in a native language such as C++ and using COM directly. Unfortunately, we were not aware of many of these performance pitfalls until after a substantial amount of code was written.

4.3.2.2 Grid Preprocessing

One downside to the BinaryMinEntropyTree algorithm described in Chapter 4.2 is that it can take a long time on large spreadsheets. While spreadsheets rarely approach the maximum size supported in Microsoft Excel (16 thousand columns by 1 million rows), spreadsheets with hundreds of rows and thousands of columns are indeed commonplace. BinaryMinEntropyTree is also difficult to effectively parallelize because binary splits rarely contain equal-sized subdivisions, meaning that parallel workloads are imbalanced.

Nonetheless, one can take advantage of an idiosyncracy in the way that people construct spreadsheets to dramatically speed up this computation. It is often the case that humans use contiguous, through-spreadsheet[10] columns or rows of a single class of values as delimiters. These delimiters are usually, but not always either whitespace or string headers.

By scanning the spreadsheet for contiguous columns or rows of equal location-free fingerprints, spatio-structural analysis is able to subdivide spreadsheets into smaller pieces which are more effectively parallelized. Through-spreadsheet rows and columns *should* be clustered together, so preprocessing a spreadsheet in this manner does not

[10] "Through-spreadsheet" should be understood here to mean the portion of the spreadsheet inside the "used range" of a spreadsheet. While all spreadsheets are technically 16 thousand columns by 1 million rows, the vast majority of that space is typically whitespace. ExceLint draws the smallest bounding box around the non-whitespace values in a spreadsheet, referred to as the *used range*, and analyzes only that region.

run the risk of encouraging the binary tree decomposition algorithm to produce bad subdivisions later on because dividing a contiguous row or column tends to increase entropy. Preprocessing spreadsheets in this way also preserves the intuitive boundaries that are likely intended by programmers.

In our experiments, the effect of this optimization was dramatic: the time taken for large spreadsheets, which sometimes took tens of minutes to compute, were computed in seconds after preprocessing was applied. Scanning for these splits is also inexpensive, since there are only $O(\text{width} + \text{height})$ possible splits. ExceLint uses all the splits that it can find.

4.3.2.3 Bitvector optimization

In practice, subdividing set of cells and computing their entropy is somewhat expensive. A cell address object in ExceLint stores not just information relating to its x and y coordinates, but also its worksheet, workbook, and full path on disk, and are therefore "big" objects[11]. A typical analysis stores tens or hundreds of thousands of address objects, one for each cell in an analysis. Address comparisons are frequent in ExceLint, as is checking for address equivalence (an operation that .NET's `System.Collections.HashSet<T>` invokes any time an element is added to a set, usually via the `HashCode` method defined on `T`). Furthermore, fingerprint values for addresses must be repeatedly recalled or computed and then counted to compute entropy.

Another way of storing information about the distribution of fingerprints on a worksheet uses the following implicit encoding scheme, borrowed from FlashRelate (see Chapter 2.4). For each class of location-free fingerprints on each worksheet (i.e., for each unique fingerprint), we store one bitvector. This means that, for each address, no more than f bits are stored, where f is the number of unique fingerprints.

[11] An ExceLint `Address` object contains two 32-bit integers, and three 64-bit managed references.

f is often small, so the total number of bitvectors stored is also small. A bitvector encodes whether a cell at a given location contains that fingerprint, a 1 if it does, otherwise 0. Computing entropy for a spreadsheet largely reduces to counting the number of ones present in each bitvector.

Since the rectangular decomposition algorithm needs to find entropy for subdivisions of a worksheet, masked bitvectors need to be computed. The bitvector mask corresponds to the region of interest, where 1 represents a value inside the region and 0 represents a value outside the region. Bitwise `AND` of the fingerprint bitvector and the mask yields a bitvector where 1 is an instance of the fingerprint inside the region, otherwise 0. The entropy of subdivisions can then be computed the same way as mentioned above, since all instances of a fingerprint appearing outside the region of interest appear as 0 in the subdivided bitvector.

The following bijective function maps (x, y) coordinates to a bitvector index.

$$\text{Index}_s(x, y) = (y - 1) \cdot w_s + x - 1$$

where w_s is the width of worksheet s. The relation subtracts one from the result because bitvector indices range over $0 \ldots n - 1$ while address coordinates range over $1 \ldots n$.

As with FlashRelate, the number of bits set can be counted in $O(b)$ time, where b is the number of bits set [90]. Since the time cost of setting bits is $O(b)$ and bitwise `AND` is $O(1)$, the total time complexity is $O(f \cdot b)$, where f is the number of fingerprints on a worksheet[12]. As with grid preprocessing, counting this way dramatically sped up the analysis, roughly by a factor of 4.

[12]Technically, bitvector `AND` takes amortized contstant time, since ExceLint uses .NET's `BigInteger` class. `BigInteger` is encoded as a variable-length bitvector.

4.4 Evaluation

The evaluation of ExceLint focuses on answering the following research questions:

1. Does the ExceLint regularity map visualization find new bugs?
2. Is the ExceLint proposed fix tool precise?
3. How does ExceLint compare against state-of-the-art pattern-based bug finders?
4. Is ExceLint fast enough to use in practice?

4.4.1 Goals

An important goal of ExceLint is to favor precision over recall, based on the observation that low-precision, high-recall tools are usually perceived by users as untrustworthy [14]. Interactivity is also an important goal, and drove much of ExceLint's algorithmic development to favor low-latency approaches. In other words, analyses need to be both fast and precise. Finally, unlike other tools, we built a real graphical user interface for ExceLint as a plugin for Microsoft Excel. The purpose of the research questions above are to evaluate ExceLint on all the dimensions we think are important: (1) bug finding ability, (2) time savings, and (3) usability.

4.4.2 Result Summary

The ExceLint regularity map visualization helps uncover many new reference bugs when used on an existing pre-audited corpus. When using ExceLint to propose fixes, ExceLint is more precise than a comparable state-of-the-art smell-based tool. Finally, ExceLint is fast enough to run interactively, requiring a median of 7.8 seconds to run a complete analysis on an entire workbook.

On an important note, the experience of building a real user interface had a strong effect on the methods used to report bugs to users. While we initially produced a tool that could pinpoint fixes with high precision, we found that we had a strong preference for ExceLint's regularity map. The regularity map allows users to quickly

locate anomalous references while graphically providing enough context to answer the question of whether an anomaly is a bug. While using the proposed fix tool, we found that we repeatedly referred back to the regularity map to uncover the "big picture" about the prevailing reference patterns in a spreadsheet. This observation led us to change the proposed fix tool to highlight both the error and the paired invariant cluster. This change made the proposed fix much more useful, although we found that it still did not completely supplant the regularity map. In fact, we now view the two tools as complementary.

4.4.3 Evaluation Platform

ExceLint was evaluated on typical software development hardware, a 2017 Apple MacBook Pro with a 1TB solid state disk, 16GB of RAM, and a two-core 3.5GHz Intel Core i7. Since ExceLint is written using a plugin framework only available on Microsoft Windows, ExceLint was run inside the 64-bit version of VirtualBox (version 5.1.26) using Microsoft Windows 10 (version 1607) with 8GB of RAM. ExceLint's cluster analysis is written using multithreaded code. Since the cluster analysis time generally dominates the analysis time, users can expect speedups roughly proportional to the number of cores.

4.4.4 RQ1: Does ExceLint's Regularity Map Find New Bugs?

In order to evaluate whether ExceLint's regularity map visualization finds new bugs, we used a set of 68 annotated spreadsheets drawn from the EUSES corpus [42]. These spreadsheets were annotated by researchers investigating spreadsheet smells using a tool called CUSTODES [24]. While the purposes of ExceLint and CUSTODES are different—the former finds reference bugs and the latter finds spreadsheet smells—many cells are considered both reference bugs and "smelly." Re-auditing the same corpus with a different tool helps establish whether ExceLint helps uncover

reference bugs that CUSTODES does not find. We also use the new annotated corpus when evaluating questions in Chapters 4.4.5 and 4.4.6.

4.4.4.1 Procedure

We annotated 29 of the 68 spreadsheets provided by CUSTODES researchers. The annotation procedure was as follows. Spreadsheets were chosen from the CUSTODES corpus in alphabetical order. Each spreadsheet was opened and the ExceLint regularity map was displayed. Regions that contained visual anomalies were inspected either by clicking on the formula and examining its references, or by using Excel's formula view. If the cell was found to violate a reference invariant, it was labeled as a reference bug. In cases where a bug had a clear bug dual (see Chapter 4.1.3), both sets of cells were labeled, and a note was added that one set was the dual of the other. After running ExceLint, we then inspected the CUSTODES ground truth for the same spreadsheet, following the same procedure as before, with one exception. If it was clear that a labeled smell was clearly not a reference bug, it was labeled "not a bug" in our corpus.

4.4.4.1.1 Counting Reference Bugs. The presence of bug duals complicates counting the "true number of bugs." In the absence of ground truth, it is impossible to determine mechanically which set—anomalous or non-anomalous—represents the true bug. Worse, even in the presence of ground truth, which we hand-curated for this work, the true bug is often ambiguous, and requires knowing the user's intent. The intent of spreadsheet programs written using insider jargon, such as in finance, or using foreign languages can be difficult to decipher without missing context. Nonetheless, it is often quite clear when reference errors occur, because reference invariants are broken. This fact is analogous to off-by-one errors in an ordinary programming language; it is not necessary to know what the user intended to accomplish to be able to observe that the code is flawed. We were often surprised at the ease of finding reference errors purely

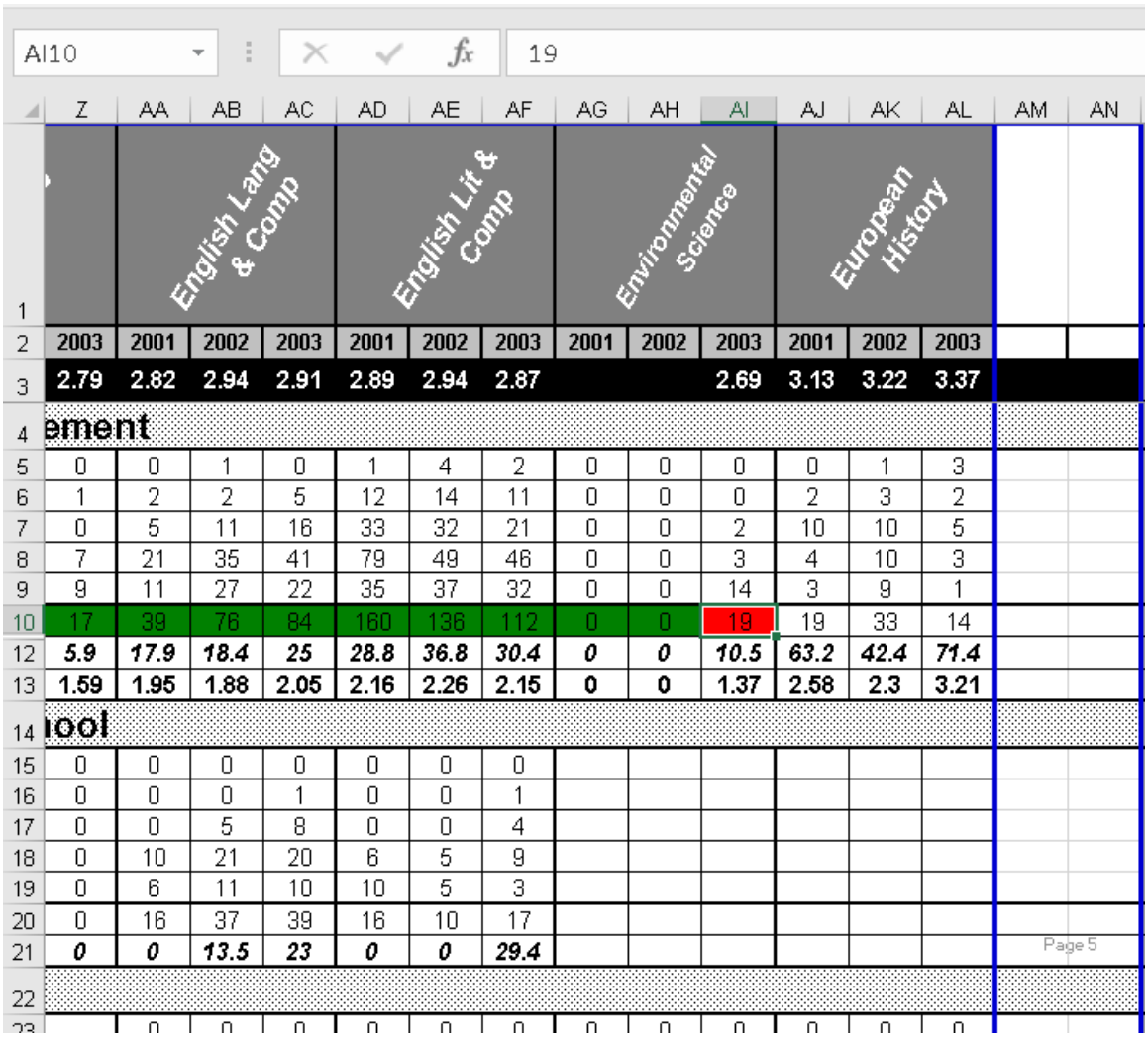


Figure 4.15: ExceLint's bug finder visualization. Cells that maintain the mined invariant are highlighted in green and the cell that violates the invariant is highlighted in red. Here, cell `AI10` is a hard-coded constant where the formulas in green, `C10:AH10`, are summary formulas. Since `AI10` is the one and only reference bug in this row, we count at most one true postive for any cells flagged in `C10:AI10`. Note that formulas `AJ10:AL10` also maintain the same reference invariant as `C10:AH10`, so it is clear upon manual inspection that `AI10` is a latent error (see Chapter 1.3.1).

by reasoning about reference invariants, even when the purpose of the calculation was unknown.

Nonetheless, we still must produce meaningful bug counts in order to evaluate ExceLint. We use the following scheme. Let a cell flagged by a bug-reporting tool be called a *flagged cell*. If a flagged cell is not a bug, we add nothing to the total bug

count. If a flagged cell is a real bug, but has no dual, we add one to the total bug count. If a flagged cell is a bug, and has a bug dual, then we maintain a count of all the cells flagged for the given dual. The maximum number of bugs added to the total bug count is the number of cells flagged from either set in the dual, or the size of the anomalous set, whichever is smaller. This scheme was chosen because, in the case where an anomaly is correct, finding a broken invariant often reveals large swaths of incorrect cells. Yet after identifying a broken invariant, the marginal effort required to flag the large number of actually incorrect cells is generally very small. Counting bug duals by the size of the smaller set is therefore a more accurate measure of the actual *effort* needed to audit using anomalies. This fact also renders moot the question of which of the two sets of a bug dual is the actual bug.

Figure 4.15 shows the case where an anomaly is a true positive and we therefore would count only a single bug. Figures 4.22 and 4.23 shows the case where the anomalous formulas are correct but the larger invariant cluster is wrong. In this latter case, we would count up to four bugs (the size of the anomalous set), even if we flagged all of the cells in the larger cluster (which are all incorrect).

4.4.4.2 Results

For the 29 spreadsheets we annotated, the CUSTODES ground truth file indicates that 1199 cells are smells. Our audit shows this data actually flags 841 reference bugs that also happen to be smells. Using ExceLint, we found an additional 1658 reference bugs, for a total of 2499 reference bugs. We spent roughly 17 hours auditing these spreadsheets. Since we did not perform an unassisted audit for comparison, we do not know how much time we saved versus not using the tool. Nonetheless, since an unassisted audit would require examining *all* of the cells individually, the savings are likely substantial. On average, we uncovered 2.45 reference bugs every minute, which is arguably an effective use of auditor effort.

4.4.5 RQ2: Is the ExceLint Proposed Fix Tool Precise?

While the regularity map is an effective tool for finding bugs, it complicates direct comparisons with other bug-finding tools, since it does not identify which cells are likely bugs. By contrast, the proposed fix tool is more amenable to direct comparisons. As mentioned earlier, we often found ourselves referring back to the regularity map when using the proposed fix tool, so we regard it as more of a complimentary tool than a replacement for the regularity map. Nonetheless, a comparison is informative.

4.4.5.1 Procedure

As described in Chapter 4.3.1.2, ExceLint's proposed fix tool highlights two things when the user requests an audit: (1) a set of suspect anomalies and (2) the corresponding set of non-anomalies. In most cases, the set of suspect anomalies contains a single cell, but it may contain as many cells as the corresponding set of non-anomalies.

To determine ExceLint's precision for true reference bugs, a highlighted cell uncovers a true bug if either (1) the flagged cell is either marked as an anomaly in the ground truth data or (2) it is an anomaly dual in the ground truth data. We count the number of true positives using the procedure described earlier (see "Counting Reference Bugs"). When ExceLint flags nothing, we adopt the standard convention of defining precision to be 1, since it makes no mistakes, even though recall in this case is 0.

4.4.5.2 Results

On average, ExceLint flags more than 3 out of every 4 cells correctly. ExceLint has a mean precision of 0.766. As we discuss in the introduction to this evaluation, usability concerns biased us strongly in favor of a high-precision tool instead of a tool with high recall. ExceLint's recall is a correspondingly low 0.198. ExceLint flags a median of 3 true positives, and a median of 0 false positives.

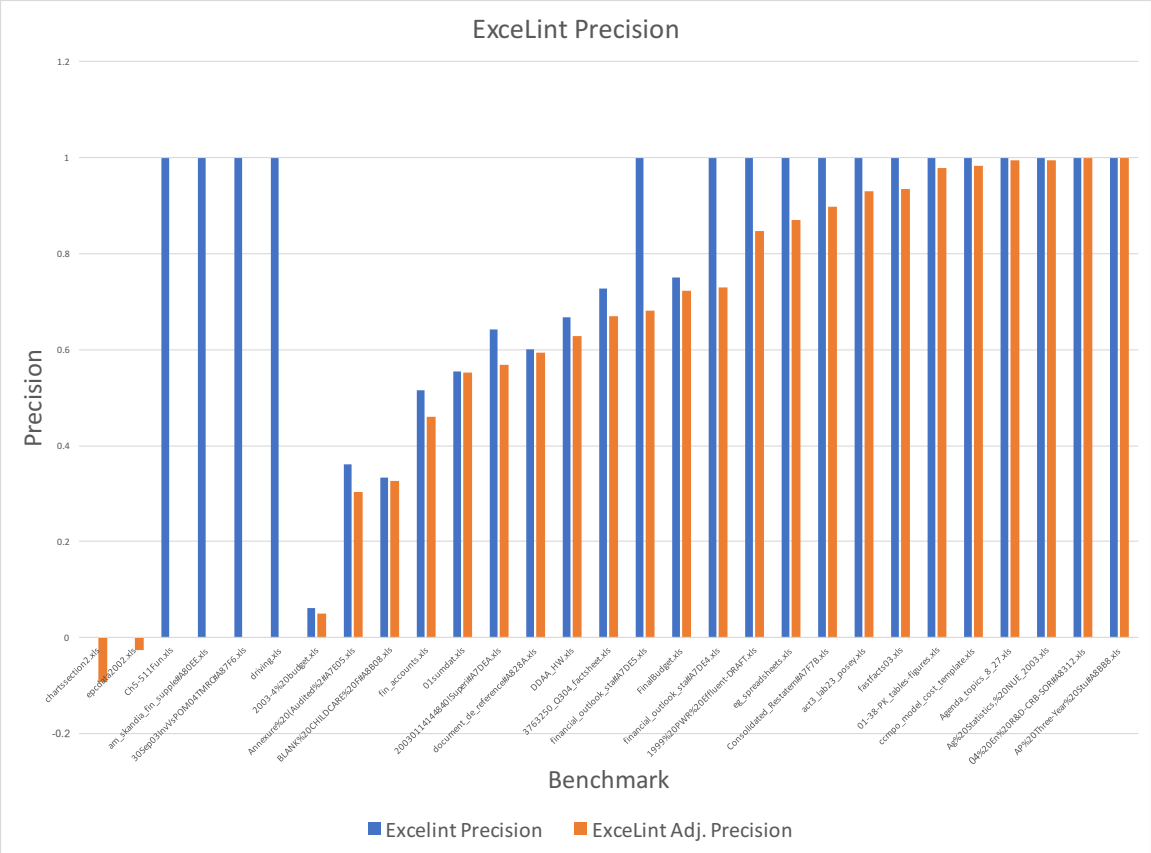


Figure 4.16: ExceLint's precision finding reference bugs across the CUSTODES benchmark suite. Results are sorted by adjusted precision (see Chapter 4.4.5).

Because it is possible that ExceLint achieves high precision because it either (1) flags nothing, or because (2) there are so many bugs in a spreadsheet that random-flagging is a good strategy, we also compute an adjusted precision that takes into account the expected precision obtained by a random flagging tool. Since spreadsheets are relatively small, we use the expected value for the hypergeometric distribution, which is defined as

$$\mathbb{E}[X] = n\frac{r}{m}$$

where X is a random variable representing the number of true positives, m is the total number of cells in the "used range" of the workbook, r is the set of true reference

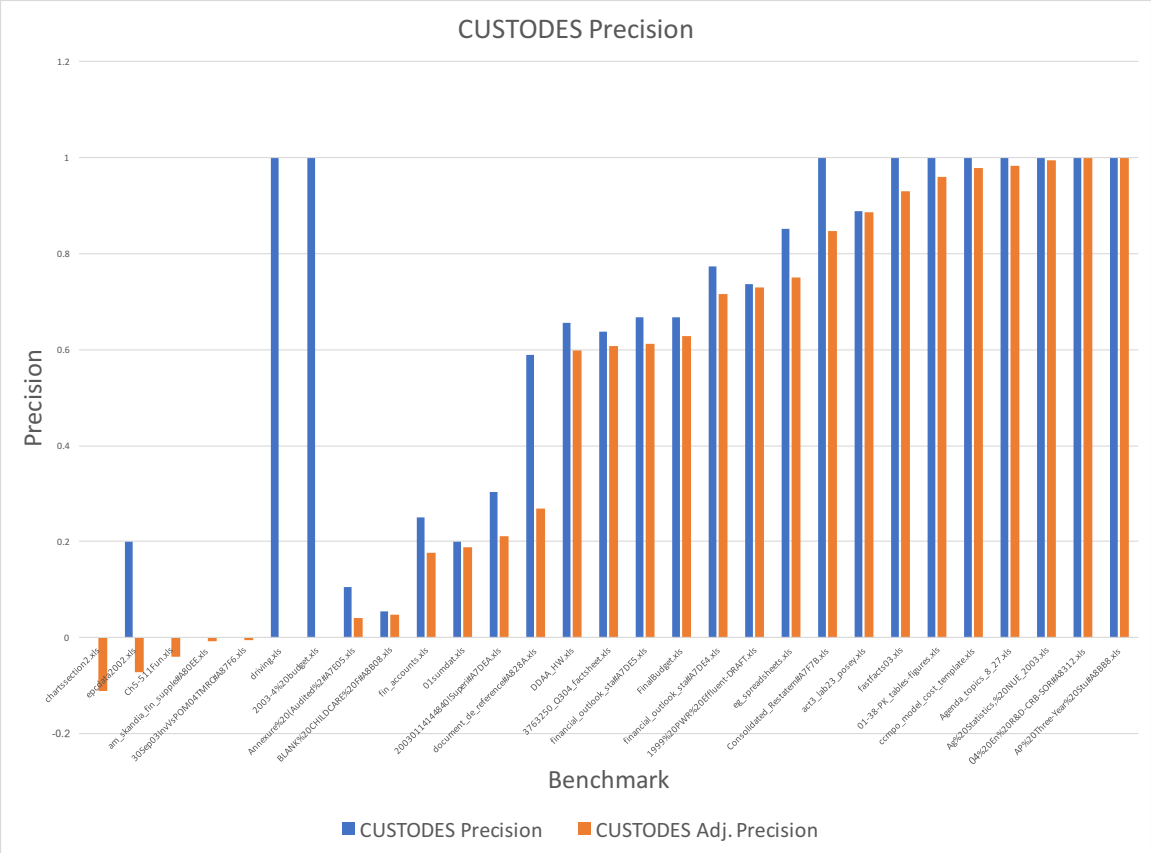


Figure 4.17: CUSTODES' precision finding reference bugs across the CUSTODES benchmark suite. Results are sorted by adjusted precision (see Chapter 4.4.6).

bugs in the workbook according to the ground truth, and n is the size of the sample, i.e., the number of bugs requested by the user. We fix n to be the same number of cells as flagged by ExceLint.

We define TP_a, the adjusted number of true positives, to be $TP - \mathbb{E}[X]$. We define FP_a, the adjusted number of false positives, to be $TP + FP - TP_a$. Correspondingly, we define the adjusted precision to be 0 when both the random flagger and ExceLint flag nothing, and $\frac{TP_a}{TP_a+FP_a}$ otherwise.

The random number of true positives $\mathbb{E}[X]$ for this corpus ranges from 0 to 10.8. In only 2 of the 29 cases, the random flagger performs better than ExceLint. For benchmarks where ExceLint achieves perfect precision because it flags nothing, a random flagger does equally well, so the adjusted precision is 0 (4 cases). Finally, when

benchmarks are small, or have a large number of true positives, ExceLint's adjusted precision is lower than its raw precision. ExceLint's mean adjusted precision is 0.572. In general, this suggests that ExceLint's performance is not strongly dictated by either poor benchmarks or by the cases where it does nothing.

A table comparing ExceLint's and CUSTODES' complete reference bug-finding results is shown in Table 4.1 and charts are shown in Figures 4.16 and 4.17.

4.4.5.2.1 Negative Expected Precision. Some discussion of ExceLint's 2 cases where the random flagger does better is in order since it reveals some limitations of an anomaly-based approach.

In the first case, `chartssection2.xls`, ExceLint flags a number of formulas of the form `=[constant`$_1$`] +...+ [constant`$_n$`]` where n varies considerably. Many of these cells have differing numbers of constants. Nonetheless, despite being clearly suspicious with respect to their surrounding cells (which are just data), it is not clear that they are erroneous, so we marked them as "not a bug" during our audit. As a result, ExceLint flags no true positives in this spreadsheet. The known true reference bugs in this worksheet occur in the summary rows that appear at the bottom of each table. Unfortunately, so few of these rows exhibit a consistent pattern that ExceLint does not learn any meaningful invariant and it does not flag them. However, a large number of totals rows are clearly wrong upon inspection, which is why the random flagger performs better.

In the second case, `epcdata2002.xls`, two kinds of reference bug dominate: constants where formulas are expected, and summary rows that sum over large swaths of non-numeric data, such as whitespace and headers. Again, ExceLint flags no true positives on this spreadsheet. For the first kind of bug, ExceLint does not flag these cells because, while it is apparent as a human that a formula is missing, there are no nearby invariant clusters for ExceLint to mine. For the second kind of bug, erroneous summary rows are wrong without any deviation. While ExceLint mines an

invariant, this invariant actually describes an entire bad neighborhood. Since there are large numbers of these two kinds of bug, the random flagger outperforms ExceLint. The cells ExceLint does flag do indeed violate local reference invariants, however, the creator of the spreadsheet entered comments and other contextual information to signal that these violations were intentional. Therefore, we marked them as "not a bug" during our manual audit.

4.4.5.3 Summary

These results suggest that ExceLint's proposed fix tool is effective at helping users identify real reference bugs. ExceLint's median of 3 true positives and 0 false positives suggest that ExceLint also does not incur much auditing effort on the part of users.

4.4.6 RQ3: How Does ExceLint Compare Against CUSTODES?

The previous procedure allow a direct comparison betweem ExceLint and CUSTODES. We measure CUSTODES' performance against the true reference identified in our audit. For completeness, we also evaluate ExceLint's and CUSTODES' peformance against smells.

4.4.6.1 Procedure

To evaluate CUSTODES performance finding true reference bugs, we use the same procedure as outlined above except that we use the set of cells identified as smells by the CUSTODES tool. Note that in order to do this comparison, we needed to build an ExceLint-like user interface for CUSTODES, which is a command-line tool that produces no visualizations.

We also compare ExceLint using CUSTODES' own criteria: a flagged cell is a "true smell" if the ground truth data marks the cell as "smelly." Note that CUSTODES has no corresponding notion of duals (see Chapter 4.1.3) for smells, so

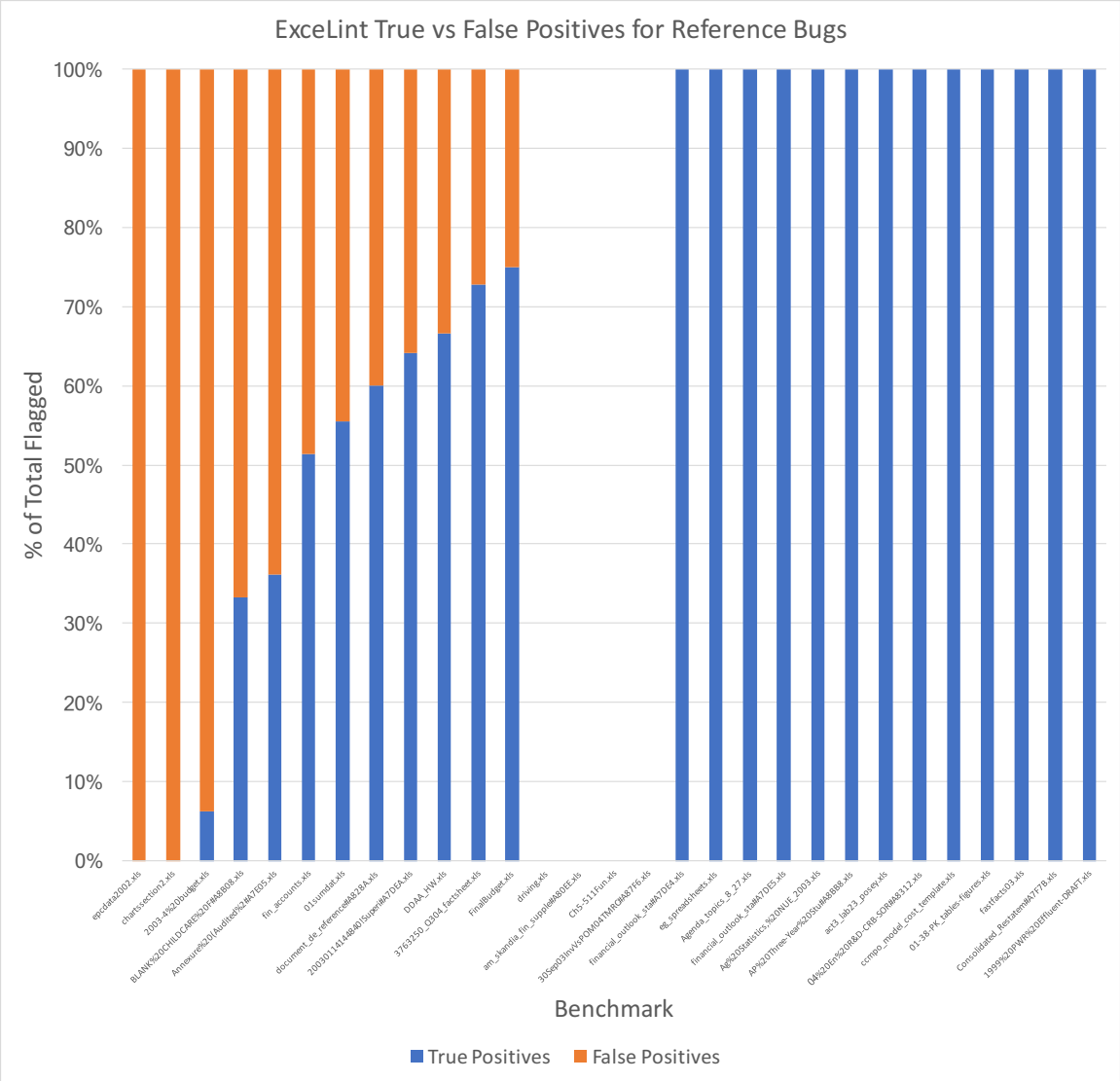


Figure 4.18: ExceLint's mix of true and false positives finding reference bugs for the CUSTODES suite (see Chapter 4.4.5).

there is a one-to-one correspondence between the number of correctly flagged items and the number of true positives.

4.4.6.2 Results

CUSTODES mean precision when locating true reference bugs is 0.606, which is lower than ExceLint's mean precision of 0.766. CUSTODES has a higher mean

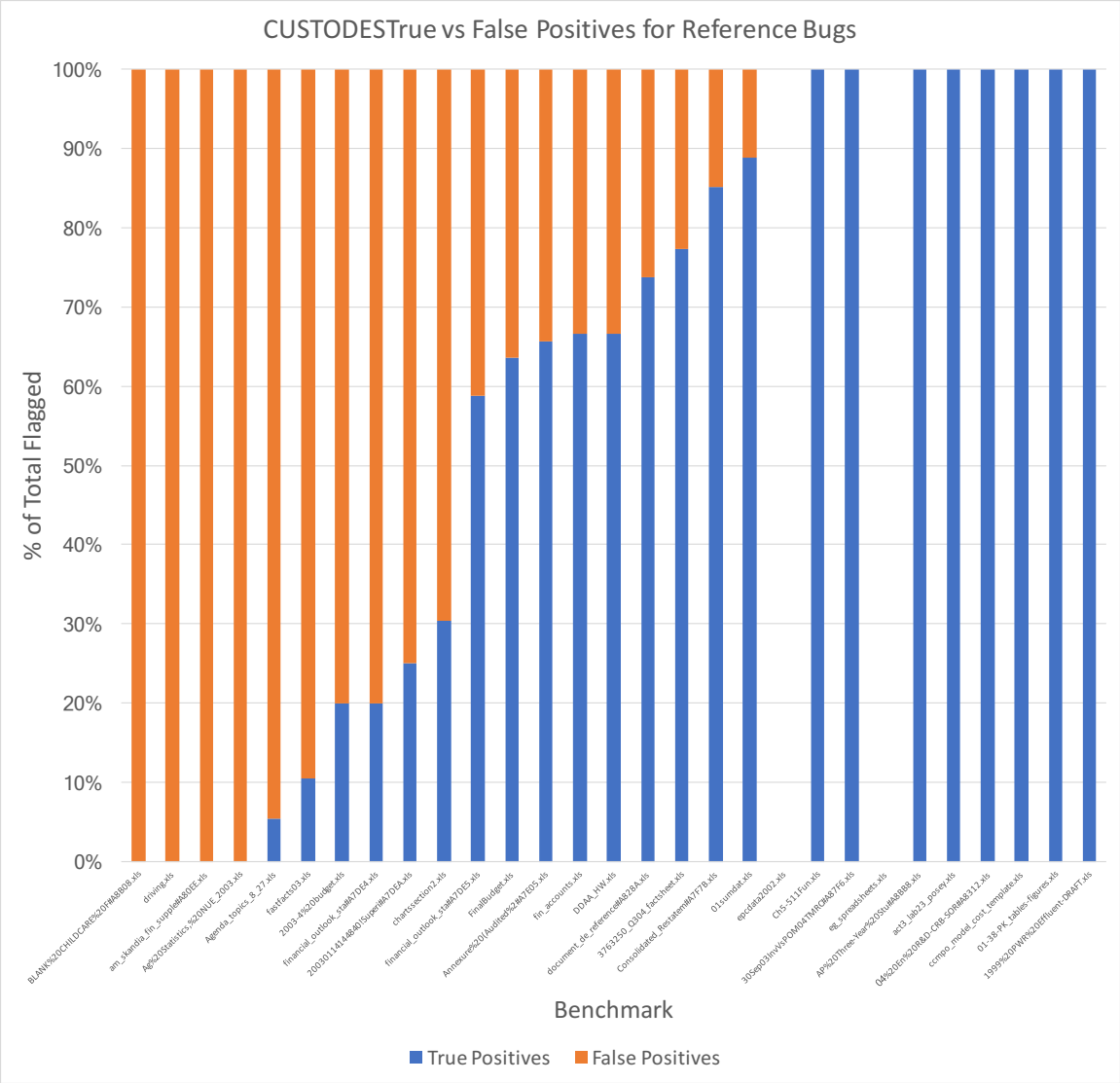


Figure 4.19: CUSTODES's mix of true and false positives finding reference bugs for the CUSTODES suite (see Chapter 4.4.6).

recall than ExceLint with a value of 0.315 vs ExceLint's 0.198[13]. Using the same procedure to compute a random-flagging baseline, except that we use the number of cells that CUSTODES flags, CUSTODES adjusted mean precision is 0.480 versus ExceLint's 0.572. The expected number of true positives for a random flagger

[13]Boosting recall over their previous tool, AmCheck, was an explicit goal of the CUSTODES work [91].

ranged from 0 (because it flags nothing) to 77.4. In 5 cases, the random flagger outperformed CUSTODES. In 2 cases, CUSTODES flagged nothing. CUSTODES flags a median of 3 true positives and 0.305 false positives.

With regard to finding smells, the tables are mostly turned. ExceLint's mean precision finding smells is 0.589 and its recall is 0.305. Not surprisingly, CUSTODES performance in smell detection is better, with a mean precision of 0.736 and a mean recall of 0.660.

A table comparing ExceLint's and CUSTODES' complete smell-finding results is shown in Table 4.2. Plots showing ExceLint's and CUSTODES' mix of true and false positives for true reference bugs are shown in Figures 4.18 and 4.19.

4.4.6.3 Summary

ExceLint outperforms CUSTODES when finding reference bugs. Both ExceLint's raw and adjusted mean precision values of 0.766 and 0.572, respectively, dominate CUSTODES' raw and adjusted values of 0.606 and 0.480. Furthermore, CUSTODES's higher median false positives suggest that CUSTODES' imposes a higher burden on users than ExceLint, since it flags true positives and false positives in roughly equal proportion.

CUSTODES outperforms ExceLint when finding smells. This is not surprising, since ExceLint is not designed to find smells. While smells may indeed signal problems with spreadsheet code, determining whether cells flagged as smelly are incorrect requires additional effort. Reference bugs are indisputably deviations in reference behavior, so we think that ExceLint is better suited for finding real bugs in spreadsheets. An important contribution of this work is a more precise definition of the kinds of features that matter in spreadsheets. In this regard, spatio-structural vectors are clearly better suited for finding reference errors.

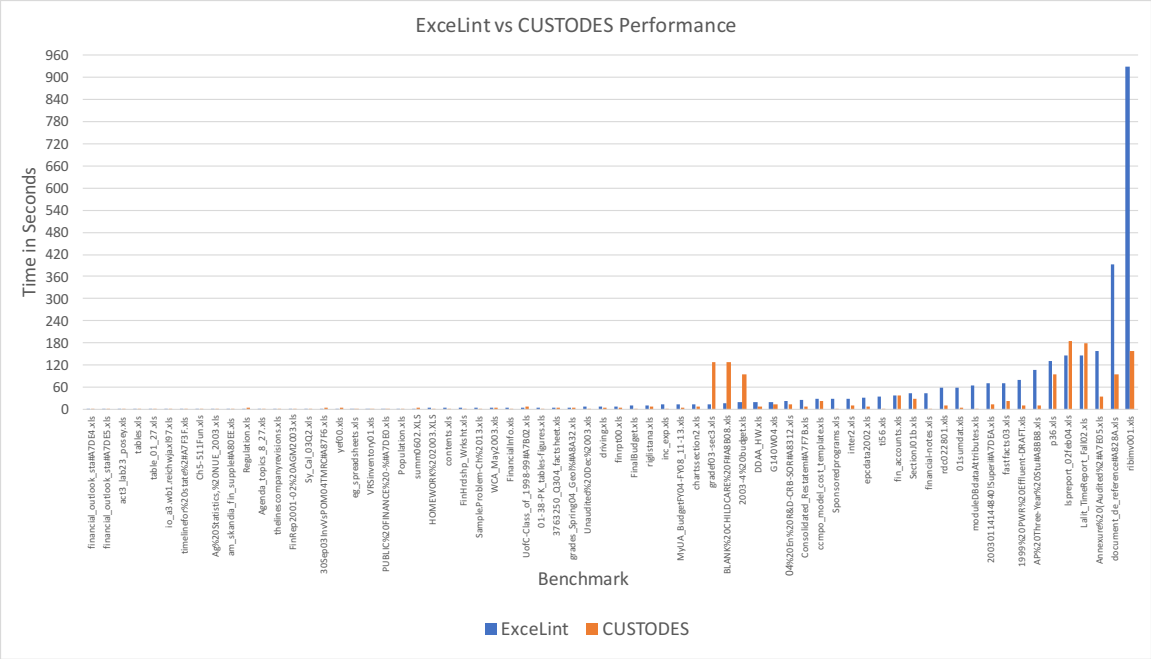


Figure 4.20: ExceLint vs CUSTODES run times. ExceLint and CUSTODES have similar performance, typically requiring a few seconds to run an analysis, although CUSTODES maintains an edge.

Finally, it should be noted that unlike ExceLint, CUSTODES provides few affordances for non-programmers. CUSTODES is a Java program, and takes a spreadsheet as input using the command line. As output, it prints an unranked set of cell addresses to inspect. By contrast, ExceLint is a plugin for Microsoft Excel, provides two visualizations, including a proposed fix tool that *ranks* bugs in order of their likelihood to be bugs, and is readily usable by non-programmers, who only need to click the audit buttons on a toolbar.

4.4.7 RQ4: Is ExceLint Fast?

ExceLint is fast enough to be used in an interactive fashion. While ExceLint's mean runtime is 44.3 seconds, this figure is strongly affected by the presence of two long running outliers. ExceLint's median runtime is 7.83 seconds, and 73.5 analyses run under 30 seconds.

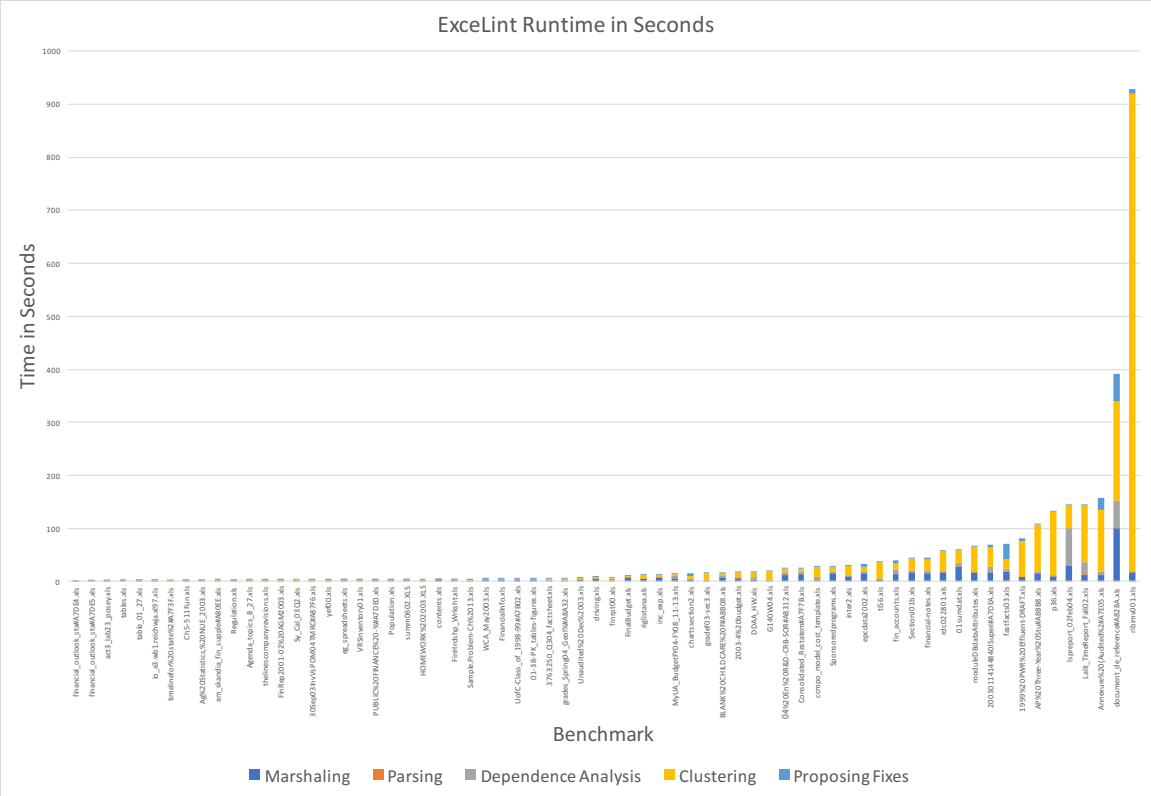


Figure 4.21: ExceLint's performance across the CUSTODES benchmark suite. Nearly 3/4 of the benchmarks run under 30 seconds for a whole-spreadsheet analysis. Most of ExceLint's analysis time is spent marshaling data from Excel.

A table comparing ExceLint's and CUSTODES' complete runtimes is shown in Table 4.3 and the same results are shown graphically in Figure 4.20. Figure 4.21 shows ExceLint's cost breakdown for each component of the analysis: marshaling, parsing, dependence analysis, clustering, and proposing fixes. Clustering time usually dominates ExceLint's analysis.

4.4.8 Case Study: The Reinhart and Rogoff Spreadsheet

We first introduced the Reinhart-Rogoff spreadsheet in Chapter 3.5.2 in the context of evaluating CheckCell's data error-finding capabilities. Since ExceLint is designed to find a complementary class of spreadsheet errors and because the

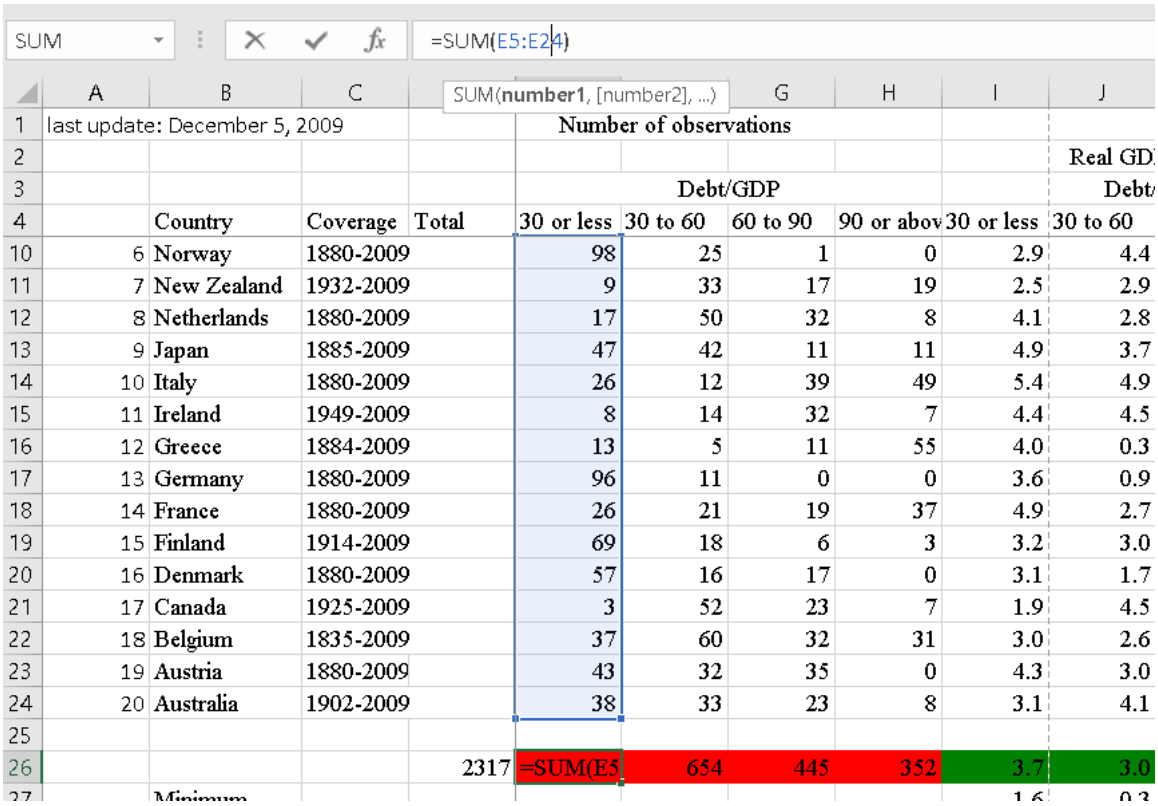
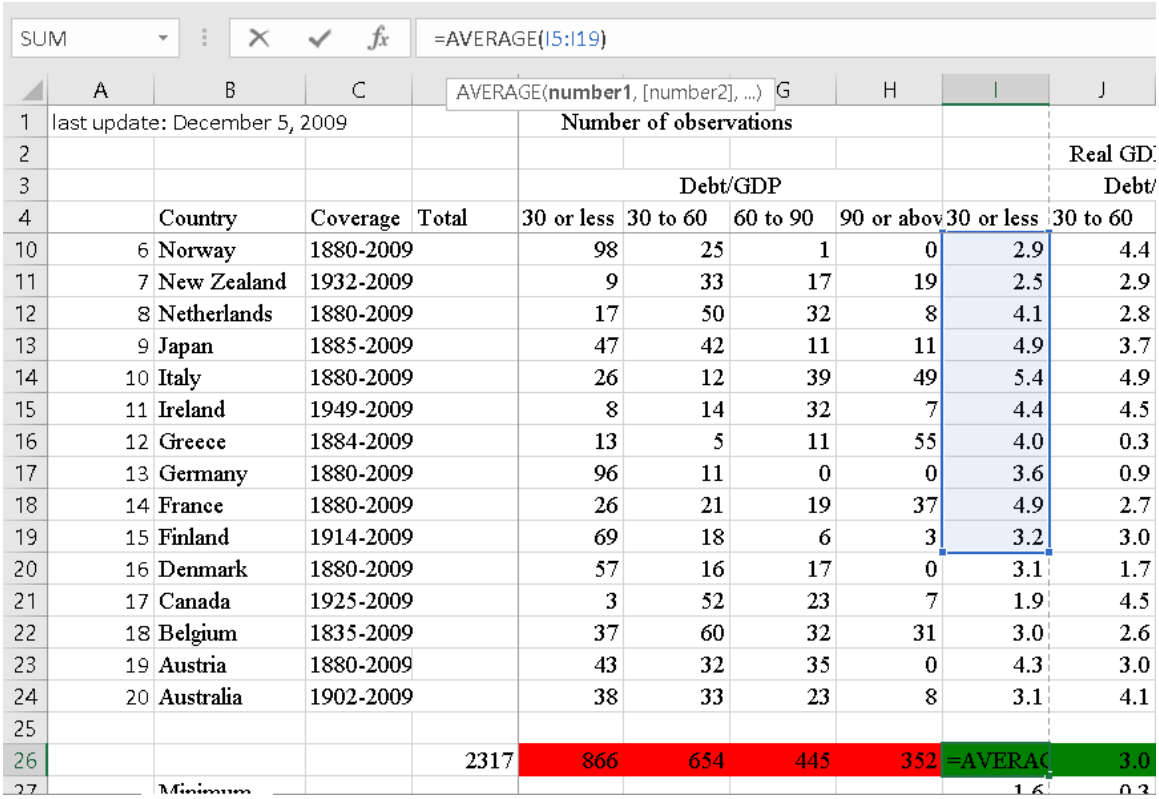


Figure 4.22: Reinhart-Rogoff reference anomalies. ExceLint flags the formulas in `E26:H26` as anomalous, breaking the reference invariant maintained by formulas in `I26:X26`. In fact, `E26:H26` are correct while all of the formulas in `I26:X26` are incorrect. Our decision to draw attention to both anomalies and non-anomalies makes the error immediately clear. In this figure, cell `E26` is highlighted to demonstrate its set of referents. We show the set of referents for cell `I26` in Figure 4.23.



	A	B	C								J
1	last update: December 5, 2009				Number of observations						
2											Real GD
3					Debt/GDP						Debt/
4		Country	Coverage	Total	30 or less	30 to 60	60 to 90	90 or abov	30 or less	30 to 60	
10	6	Norway	1880-2009		98	25	1	0	2.9	4.4	
11	7	New Zealand	1932-2009		9	33	17	19	2.5	2.9	
12	8	Netherlands	1880-2009		17	50	32	8	4.1	2.8	
13	9	Japan	1885-2009		47	42	11	11	4.9	3.7	
14	10	Italy	1880-2009		26	12	39	49	5.4	4.9	
15	11	Ireland	1949-2009		8	14	32	7	4.4	4.5	
16	12	Greece	1884-2009		13	5	11	55	4.0	0.3	
17	13	Germany	1880-2009		96	11	0	0	3.6	0.9	
18	14	France	1880-2009		26	21	19	37	4.9	2.7	
19	15	Finland	1914-2009		69	18	6	3	3.2	3.0	
20	16	Denmark	1880-2009		57	16	17	0	3.1	1.7	
21	17	Canada	1925-2009		3	52	23	7	1.9	4.5	
22	18	Belgium	1835-2009		37	60	32	31	3.0	2.6	
23	19	Austria	1880-2009		43	32	35	0	4.3	3.0	
24	20	Australia	1902-2009		38	33	23	8	3.1	4.1	
25											
26				2317	866	654	445	352	=AVERAG	3.0	
27		Minimum							1.6	0.3	

Figure 4.23: Reinhart-Rogoff reference invariant. Formula `I26` is highlighted to demonstrate its set of incorrect referents. It should refer to the same set of cells as the formula in `I26` shown in Figure 4.22.

Reinhart-Rogoff spreadsheet's errors have been thoroughly documented, we revisit that spreadsheet here.

In their analysis of the Reinhart-Rogoff spreadsheet, Herndon et al. call out one class of reference error as having a particularly significant impact on the spreadsheet [55]. In essence, the computation completely excludes five countries—Denmark, Canada, Belgium, Austria, and Australia—from the analysis:

> This spreadsheet error, compounded with other errors, is responsible for a 0.3 percentage- point error in RRs published average real GDP growth in the highest public debt/GDP category. It also overstates growth in the lowest public debt/GDP category (0 to 30 percent) by +0.1 percentage point and understates growth in the second public debt/GDP category (30 to 60 percent) by 0.2 percentage point.

Running ExceLint on the Reinhart-Rogoff spreadsheet finds this error repeated in two places on their summary sheet. Figures 4.22 and 4.23 show one of the sets of errors. The cells in red, `E26:H26`, denote the set of anomalous cells and the cells in green, `I26:X26`, denote the set of cells that maintains the inferred invariant. In fact, `I26:X26` is wrong and `E26:H26` is correct, because `I26:X26` fails to refer to the entire set of figures for each country, the cells in rows 5 through 24. Nonetheless, by highlighting both the anomaly and the invariant, it is immediately clear which of the two sets of cells is wrong. This case study justifies our decision to highlight both sets of cells, since it is often difficult to know without extra context which invariant is the correct one.

During the development of ExceLint, we frequently found that failures to maintain reference invariants were more important for finding reference bugs than deviations in formula expressions. The Reinhart-Rogoff spreadsheet supports our decision to make reference vectors computation-agnostic. The formulas in cells `E26:H26` compute a sum whereas the cells in `I26:X26` compute the arithmetic mean. Were ExceLint sensitive to the formula expressions themselves, we would not have found these severe errors.

		ExceLint				CUSTODES			
Benchmark	# Bugs	Precision	Recall	Expected TP	Adj. Precision	Precision	Recall	Expected TP	Adj. Precision
01-38-PK_tables-figures.xls	33	1.000	0.242	0.172	0.979	1.000	0.939	0.666	0.979
01sumdat.xls	17	0.556	0.294	0.034	0.552	0.889	0.471	0.034	0.885
04%20En%20R&D-CRB-SOR#A8312.xls	3	1.000	1.000	0.004	0.999	1.000	1.000	0.004	0.999
1999%20PWR%20Effluent-DRAFT.xls	510	1.000	0.139	10.819	0.848	1.000	0.996	77.407	0.848
2003-4%20budget.xls	17	0.063	0.059	0.196	0.050	0.200	0.118	0.123	0.188
20030114144840!Superi#A7DEA.xls	173	0.641	0.145	2.850	0.568	0.250	0.012	0.585	0.177
30Sep03InvVsPOM04TMRC#A87F6.xls	3	1.000	0.000	0.000	0.000	1.000	1.000	0.017	0.994
3763250_Q304_factsheet.xls	56	0.727	0.143	0.640	0.669	0.773	0.304	1.279	0.715
act3_lab23_posey.xls	7	1.000	0.429	0.212	0.929	1.000	0.286	0.141	0.929
Ag%20Statistics,%20NUE_2003.xls	3	1.000	0.667	0.011	0.995	0.000	0.000	0.090	-0.005
Agenda_topics_8_27.xls	3	1.000	0.333	0.006	0.994	0.055	1.000	0.342	0.048
am_skandia_fin_supple#A80EE.xls	22	1.000	0.000	0.000	0.000	0.000	0.000	0.040	-0.040
Annexure%20(Audited%2#A7E05.xls	249	0.362	0.068	2.752	0.303	0.656	0.084	1.874	0.598
AP%20Three-Year%20Stu#A8BB8.xls	3	1.000	0.667	0.002	0.999	1.000	1.000	0.003	0.999
BLANK%20CHILDCARE%20F#A8B08.xls	9	0.333	0.111	0.021	0.326	0.000	0.000	0.021	-0.007
ccmpo_model_cost_template.xls	7	1.000	0.714	0.090	0.982	1.000	0.429	0.054	0.982
Ch5-511Fun.xls	9	1.000	0.000	0.000	0.000	1.000	0.111	0.041	0.959
chartssection2.xls	107	0.000	0.000	0.833	-0.093	0.304	0.196	6.387	0.212
Consolidated_Restatem#A7F7B.xls	243	1.000	0.058	1.420	0.899	0.852	0.189	5.477	0.750
DDAA_HW.xls	140	0.667	0.029	0.232	0.628	0.667	0.043	0.348	0.628
document_de_reference#A828A.xls	82	0.600	0.329	0.304	0.593	0.737	0.341	0.257	0.730
driving.xls	48	1.000	0.000	0.000	0.000	0.000	0.000	0.776	-0.111
eg_spreadsheets.xls	19	1.000	0.053	0.131	0.869	1.000	0.000	0.000	0.000
epcdata2002.xls	83	0.000	0.000	0.161	-0.027	1.000	0.000	0.000	0.000
fastfacts03.xls	338	1.000	0.038	0.837	0.936	0.105	0.006	1.223	0.041
fin_accounts.xls	204	0.514	0.088	1.902	0.460	0.667	0.108	1.793	0.612
FinalBudget.xls	34	0.750	0.088	0.113	0.722	0.636	0.206	0.311	0.608
financial_outlook_sta#A7DE4.xls	32	1.000	0.031	0.271	0.729	0.200	0.063	2.712	-0.071
financial_outlook_sta#A7DE5.xls	45	1.000	0.022	0.319	0.681	0.588	0.222	5.426	0.269
MEAN	86.1	0.766	0.198	0.839	0.572	0.606	0.315	3.704	0.480

Table 4.1: ExceLint and CUSTODES precision for reference bugs. ExceLint has higher precision than CUSTODES when finding reference bugs. Each benchmark was annotated with true reference bugs, and corresponds with a benchmark drawn from CUSTODES' annotated corpus [24]. See Sections 4.4.5 and 4.4.6.

4.5 Conclusion

Spreadsheets are popular, often used to inform major decisions, and error-prone. This chapter introduced spatio-structural analysis, a form of static analysis that specifically targets errors in spreadsheet formulas. Spatio-structural analysis unifies spatial relationships and dependence relationships, enabling anomaly detection in the context of spreadsheets. This statistical approach avoids the need for ad hoc patterns while maintaining a low false positive rate. Our prototype spatio-structural analysis tool, ExceLint, is efficient and accurate. Despite the fact that, like any automatic tool, ExceLint cannot infer human intent, its analysis approach is effective at pinpointing actual, previously-unknown spreadsheet errors. Using the visualization, we found 2499 reference bugs, three times more than the state-of-the-art tool, CUSTODES, with a high precision on average, 0.766.

		ExceLint		CUSTODES	
Benchmark	# Smells	Precision	Recall	Precision	Recall
01-38-PK_tables-figures.xls	59	0.750	0.102	0.967	0.983
01sumdat.xls	12	0.222	0.167	0.778	0.583
04%20En%20R&D-CRB-SOR#A8312.xls	5	0.000	0.000	0.000	0.000
1999%20PWR%20Effluent-DRAFT.xls	543	1.000	0.131	1.000	0.943
2003-4%20budget.xls	16	0.160	0.250	0.700	0.438
20030114144840!Superi#A7DEA.xls	26	0.231	0.346	0.250	0.077
30Sep03InvVsPOM04TMRC#A87F6.xls	3	1.000	0.000	1.000	1.000
3763250_Q304_factsheet.xls	25	0.727	0.320	0.652	0.600
act3_lab23_posey.xls	3	1.000	1.000	1.000	0.667
Ag%20Statistics,%20NUE_2003.xls	0	0.000	1.000	0.000	1.000
Agenda_topics_8_27.xls	1	1.000	1.000	0.018	1.000
am_skandia_fin_supple#A80EE.xls	2	1.000	0.000	1.000	0.500
Annexure%20(Audited%2#A7E05.xls	43	0.191	0.209	0.500	0.372
AP%20Three-Year%20Stu#A8BB8.xls	3	1.000	0.667	1.000	1.000
BLANK%20CHILDCARE%20F#A8B08.xls	17	0.667	0.118	0.667	0.118
ccmpo_model_cost_template.xls	7	1.000	0.714	1.000	0.429
Ch5-511Fun.xls	1	1.000	0.000	1.000	1.000
chartssection2.xls	89	0.000	0.000	0.688	0.719
Consolidated_Restatem#A7F7B.xls	84	0.786	0.131	1.000	0.905
contents.xls	0	1.000	1.000	1.000	1.000
DDAA_HW.xls	13	0.500	0.231	0.667	0.462
document_de_reference#A828A.xls	49	0.556	0.510	0.894	0.857
driving.xls	5	1.000	0.000	0.714	1.000
eg_spreadsheets.xls	0	0.000	1.000	1.000	1.000
epcdata2002.xls	8	0.833	0.625	1.000	0.000
fastfacts03.xls	25	0.077	0.040	0.947	0.720
fin_accounts.xls	97	0.167	0.072	0.816	0.412
FinalBudget.xls	16	1.000	0.250	1.000	0.750
financial_outlook_sta#A7DE4.xls	15	1.000	0.067	1.000	0.667
financial_outlook_sta#A7DE5.xls	32	0.000	0.000	0.895	0.531
financial-notes.xls	41	0.250	0.049	0.773	0.829
FinancialInfo.xls	5	1.000	0.000	1.000	1.000
FinHrdshp_Wrksht.xls	2	0.000	0.000	0.333	0.500
FinRep2001-02%20AGM2003.xls	6	1.000	0.500	1.000	0.833
finrpt00.xls	11	1.000	0.000	0.900	0.818
G140W04.xls	11	0.545	0.545	1.000	1.000
gradef03-sec3.xls	0	0.000	1.000	0.000	1.000
grades_Spring04_Geol%#A8A32.xls	4	1.000	1.000	1.000	1.000
HOMEWORK%202003.XLS	0	1.000	1.000	0.000	1.000
inc_exp.xls	1	0.000	0.000	1.000	0.000
inter2.xls	422	0.000	0.000	0.961	0.924
io_a3.wb1.reichwja.xl97.xls	3	1.000	0.000	1.000	0.333
Lalit_TimeReport_Fall02.xls	6	0.200	0.167	0.833	0.833
lspreport_02feb04.xls	12	0.412	0.583	0.917	0.917
moduleDBdataAttributes.xls	18	1.000	0.000	1.000	0.611
MyUA_BudgetFY04-FY08_11-13.xls	15	0.238	0.333	0.667	0.267
p36.xls	2	1.000	0.000	1.000	1.000
Population.xls	16	1.000	0.063	1.000	0.625
PUBLIC%20FINANCE%20-%#A7DE0.xls	12	0.667	0.167	1.000	1.000
rdc022801.xls	1	0.000	0.000	0.000	0.000
Regulation.xls	22	0.800	0.364	1.000	0.045
ribimv001.xls	2	0.167	1.000	0.000	0.000
riglistana.xls	1	1.000	0.000	1.000	0.000
Sample.Problem-Ch%2013.xls	4	1.000	0.000	0.600	0.750
SectionJ01b.xls	2	0.000	0.000	1.000	0.000
Sponsoredprograms.xls	10	0.167	0.100	0.000	0.000
summ0602.XLS	0	0.000	1.000	0.000	1.000
Sy_Cal_03Q2.xls	6	1.000	0.000	1.000	1.000
table_01_27.xls	30	1.000	0.067	1.000	0.167
tables.xls	0	1.000	1.000	1.000	1.000
thelinescompanyrevisions.xls	0	0.000	1.000	0.000	1.000
ti56.xls	26	0.750	0.115	1.000	1.000
timelinefor%20state%2#A7F3F.xls	1	1.000	0.000	1.000	1.000
Unaudited%20Dec%2003.xls	7	0.000	0.000	0.857	0.857
UofC-Class_of_1998-99#A7B02.xls	7	1.000	0.143	0.000	0.000
VRSinventory01.xls	7	1.000	0.571	0.069	0.857
WCA_May2003.xls	27	1.000	0.037	0.963	0.963
yef00.xls	1	0.000	0.000	1.000	1.000
MEAN	28.529	0.589	0.305	0.736	0.660

Table 4.2: ExceLint and CUSTODES precision for smells. CUSTODES has higher precision than ExceLint when finding smells. Note that ExceLint is designed to find reference bugs, not smells. Nonetheless, the fact that ExceLint finds any smells at all suggests that many smells are reference bugs in disguise. See Sections 4.4.5 and 4.4.6.

BenchmarkName	# cells	# formulas	ExceLint (ms)	CUSTODES (ms)
01-38-PK_tables-figures.xls	1535	144	5569	2699
01sumdat.xls	4542	349	60295	5260
04%20En%20R&D-CRB-SOR#A8312.xls	2285	288	24378	12603
1999%20PWR%20Effluent-DRAFT.xls	3347	1439	81364	10470
2003-4%20budget.xls	1385	492	18626	94163
20030114144840!Superi#A7DEA.xls	2367	957	69657	14675
30Sep03InvVsPOM04TMRC#A87F6.xls	522	144	2666	3682
3763250_Q304_factsheet.xls	963	200	5672	6118
act3_lab23_posey.xls	99	40	877	1838
Ag%20Statistics,%20NUE_2003.xls	566	107	2044	2269
Agenda_topics_8_27.xls	482	3	2558	925
am_skandia_fin_supple#A80EE.xls	552	56	2489	1614
Annexure%20(Audited%2#A7E05.xls	4252	1022	157226	33592
AP%20Three-Year%20Stu#A8BB8.xls	3010	64	107641	11813
BLANK%20CHILDCARE%20F#A8B08.xls	1307	342	15967	128333
ccmpo_model_cost_template.xls	390	275	27538	23704
Ch5-511Fun.xls	221	11	1588	1152
chartssection2.xls	1156	162	14689	7432
Consolidated_Restatem#A7F7B.xls	2396	300	26263	7035
contents.xls	938	78	3903	1507
DDAA_HW.xls	3619	515	19419	7210
document_de_reference#A828A.xls	12121	2398	391585	94395
driving.xls	433	146	7720	4717
eg_spreadsheets.xls	145	47	2860	998
epcdata2002.xls	3102	581	33321	7397
fastfacts03.xls	5250	1499	71646	23265
fin_accounts.xls	3754	1027	38993	39308
FinalBudget.xls	1204	280	10908	3043
financial_outlook_sta#A7DE4.xls	118	40	452	1396
financial_outlook_sta#A7DE5.xls	141	43	742	1342
financial-notes.xls	2542	261	45061	3122
FinancialInfo.xls	461	71	4716	1544
FinHrdshp_Wrksht.xls	195	81	3974	2644
FinRep2001-02%20AGM2003.xls	348	86	2581	2156
finrpt00.xls	1049	180	7954	4181
G140W04.xls	739	293	20231	12956
gradef03-sec3.xls	1013	299	15265	129652
grades_Spring04_Geol%#A8A32.xls	431	199	5884	4699
HOMEWORK%202003.XLS	656	154	3575	3089
inc_exp.xls	614	300	12713	1337
inter2.xls	3955	44	29720	9622
io_a3.wb1.reichwja.xl97.xls	204	52	1390	1516
Lalit_TimeReport_Fall02.xls	1686	1366	145881	178170
lspreport_02feb04.xls	3520	1155	145515	184110
moduleDBdataAttributes.xls	4128	142	65269	3353
MyUA_BudgetFY04-FY08_11-13.xls	838	406	13921	3506
p36.xls	3681	558	131934	96020
Population.xls	719	9	3089	1259
PUBLIC%20FINANCE%20-%#A7DE0.xls	216	45	3055	1425
rdc022801.xls	2519	141	58552	10599
Regulation.xls	348	184	2522	4020
ribimv001.xls	5913	2662	928879	158521
riglistana.xls	1079	116	11757	7514
Sample.Problem-Ch%2013.xls	299	37	4032	1310
SectionJ01b.xls	1528	1173	43504	30402
Sponsoredprograms.xls	2654	208	29152	2280
summ0602.XLS	200	102	3298	3496
Sy_Cal_03Q2.xls	359	20	2622	1368
table_01_27.xls	192	98	1145	1415
tables.xls	290	8	899	831
thelinescompanyrevisions.xls	356	55	2562	1255
ti56.xls	767	62	36440	2856
timelinefor%20state%2#A7F3F.xls	142	11	1424	854
Unaudited%20Dec%2003.xls	492	118	6771	3100
UofC-Class_of_1998-99#A7B02.xls	556	115	4924	8397
VRSinventory01.xls	235	77	2922	2035
WCA_May2003.xls	271	111	4432	5763
yef00.xls	396	144	2823	5071
MEAN	1585.19	355.76	44397.71	21314.75
MEDIAN	753	144	7837	3851

Table 4.3: ExceLint and CUSTODES analysis run times. Speed measurements in milliseconds for ExceLint and CUSTODES to run a complete analysis for each tool. Both ExceLint and CUSTODES are generally very fast, analyzing spreadsheets on the order of seconds. See Chapter 4.4.7.

The technique presented in this dissertation shows that a simple approach can highlight significant errors in spreadsheets with speed and precision. At the same time, the current approach cannot identify other errors which are also significant, such as errors related to incorrect use of units. In the future, we plan to explore how additional dimensions of regularity, such as more higher order spatial patterns or more nuanced understanding of types, can be incorporated into our statistical analysis to increase recall while maintaining precision. We also plan to investigate performance modifications to make ExceLint's analysis incremental; such a change would amortize the costs of previous analyses by modifying the updated analysis with only new dependence information as users build their workbooks.

CHAPTER 5

FUTURE DIRECTIONS

Spreadsheets are likely to remain important for data analysis, and this dissertation addresses a number of pain points for analysts. However, a number of problems still remain.

Data integration remains a challenge. Less technical users may employ spreadsheets as a kind of lightweight data collection tool, building forms using their flexible layouts and rich markup features. Nonetheless, integrating data collected in this manner into traditional relational database systems is problematic. Since database schemas and spreadsheet forms are only loosely coupled, changes in either mean that automated data import routines will likely break [22].

Maintenance tasks are also problematic. As with expert programmers in other programming languages, building a spreadsheet is a time-consuming task, particularly for complex calculations. Users sometimes maintain spreadsheets over long periods of time, and they share them with other people [31]. Excel provides few affordances for this kind of workflow. Users new to a spreadsheet may be unfamiliar with unstated invariants, and upon maintenance, violate them, causing errors. Lightweight sanity checks, like the `assert` statement found in many programming languages, would allow spreadsheet creators to guard against many kinds of error. Additionally, spreadsheets do not lend themselves well to existing program versioning or differencing tools like `git` that simplify code reviews. Lastly, because copying and pasting is widespread, refactoring large spreadsheets is tedious in much the same way as layout transformation. Better debugging, versioning, and refactoring tools would ease the burden of creating

and maintaining correct spreadsheets, particularly if tools are designed with novice programmers in mind.

Finally, *data integrity* remains a problem. Since spreadsheets combine data and formulas in a single view, spreadsheet data must be thought of as important for program correctness as formulas. Nonetheless, when many users update a spreadsheet, data issues may creep in and be passed along. While some errors may be simple data bugs, like the kind discussed earlier, spreadsheets have little resistence to intentional tampering. Early work on data provenance for spreadsheets examines techniques for well-intentioned users, but does not consider malicious users who aim to intentionally subvert data [7].

CHAPTER 6

CONCLUSIONS

The unique combination of an expressive programming language and flexible data management ensures that spreadsheets remain one of the most popular programming environments. Spreadsheets are especially compelling when it comes to data-intensive tasks, which is why they are used widely among both expert programmers and ordinary end users. Unfortunately, spreadsheets also have shortcomings, particularly for a small set of tasks common to data analysis. This dissertation takes a holistic view of the data analysis pipeline and addresses inefficiencies that account for more than three quarters of an analyst's time.

6.0.1 Data Wrangling

Spreadsheets are unusually flexible in accepting a wide range of layout schemes, and users are able to choose the layout best suited for the task at hand. However, automated processing tools that take spreadsheets as input are generally inflexible, requiring that data adhere to predetermined forms. Users who lack the programming skills needed to convert data from one layout to another are forced to manually convert layouts in order to utilize these tools. *Layout transformation synthesis* is a program synthesis-based technique that bridges this "lock-in" problem, letting users convert layouts by providing examples of what they want.

FlashRelate implements layout transformation synthesis as a plugin for Microsoft Excel, and allows users to change their layouts using a simple point and click interface. Users refine their programs in a similar manner, providing additional examples or

marking incorrect transformations as counterexamples. FlashRelate is fast, typically taking only a few seconds on commodity hardware, and requires only a few examples from the user.

6.0.2 Data Cleaning

Bad or corrupted input data often escapes traditional techniques used to validate inputs. Furthermore, many programs exhibit subtle sensitivities to data values that pass validation checks. When these "data bugs" escape detection, spreadsheet calculations can lead to wrong decisions and large financial losses. *Data debugging* is a statistical program analysis technique that finds inputs likely to trigger program sensitivities. These inputs are either very important or they are wrong. Both cases warrant special attention.

CheckCell is an implementation of data debugging for Microsoft Excel. Users are guided through a cell-by-cell audit of the most important data values in their spreadsheet. CheckCell improves user effectiveness by focusing their auditing efforts on only those inputs most likely to cause problems. CheckCell is also fast, typically requiring only a few seconds to run an analysis on commodity hardware.

6.0.3 Formula Auditing

Large spreadsheets are likely to contain at least one formula error. Spreadsheet environments like Microsoft Excel inadvertently encourage certain classes of errors because the lack of user defined functions means that copy-and-paste workflows are common. In this scenario, users must either correctly insert address mode annotations to use Excel's automated copy-and-paste tools, or they must manually update data references by hand. Users fail at both tasks and reference bugs are common. *Spatio-structural analysis* is a program anomaly analysis-based technique that takes advantage of the spatial characteristics of spreadsheet programs in order to find reference bugs.

ExceLint is a bug finder for spreadsheets built on spatio-structural analysis, and it is available as a plugin for Microsoft Excel. Users are shown formulas that violate likely reference invariants. These violations strongly correlate with reference bugs. ExceLint is fast and effective, providing users with a high precision bug finding tool that takes only seconds to perform an analysis on commodity computer hardware.

APPENDIX A

FLARE SEMANTICS AND FLASHRELATE ALGORITHMS

$$
\begin{aligned}
\text{Cell } \gamma &: (\mathbb{N}, \mathbb{N}) \\
\text{Pair } \phi &: (\texttt{String}, \text{Cell}) \\
\text{Spreadsheet } I &: \texttt{String}^{w \times h} \\
\text{Cell list } \Gamma &: [\text{Cell}_1, \ldots, \text{Cell}_j] \\
\text{Quasi tuple } \tau &: [\text{Pair}_1, \ldots, \text{Pair}_k]
\end{aligned}
$$

Figure A.1: Types for a simple abstract spreadsheet model. These types are used by the semantics shown in Fig. A.2. γ is a cell location (a coordinate pair), ϕ is a "match" (a column name, location pair), I is an $m \times n$ matrix of Strings (the top left String is at (0,0) with the x coordinate increasing rightward and the y coordinate increasing downward), Γ is a list of locations, and τ is a tuple in list form (a list of matches).

1) $[\![\texttt{<prog>}]\!](I) = \text{Flatten}(\text{Map}(\lambda\gamma.\text{Map}(\lambda\tau.\text{ToTuple}(I,\tau), [\![\texttt{<prog>}]\!](I,\gamma)), \text{AllCells}(I)))$

2) $[\![\texttt{<cpair>}]\!](I,\gamma) = [\![\texttt{<cpair>[ ]}]\!](I,\gamma)$

3) $[\![\texttt{<spatialc><cellc>[<pseq>]}]\!](I,\gamma) = \text{Map}(\lambda\phi.\text{Map}(\lambda\tau.\phi::\tau, [\![\texttt{[<pseq>]}]\!](I,\text{snd}(\phi))), \text{Eval}(s,c,I,\gamma))$
$= \text{where } s = \texttt{<spatialc>}, \text{ and } c = \texttt{<cellc>}$

4) $[\![\texttt{[<spatialc><cellc>::<pseq>]}]\!](I,\gamma) = \text{Map}(\lambda\phi.\text{Map}(\lambda\tau.\phi::\tau, [\![\texttt{[<pseq>]}]\!](I,\gamma)), \text{Eval}(s,c,I,\gamma))$
$= \text{where } s = \texttt{<spatialc>}, \text{ and } c = \texttt{<cellc>}$

5) $[\![\texttt{[ ]}]\!](I,\gamma) = [\,]$

6) $[\![\texttt{<regx><anchor>}]\!](I,\gamma) = [\![\texttt{<regx>}]\!](\text{Value}(I,\gamma)) \wedge \exists\gamma' \in \text{Map}(\lambda\phi.\text{snd}(\phi), [\![\texttt{<anchor>}]\!](I,\gamma))$

7) $[\![\texttt{"<"<name>","<regx>">"<anchor>}]\!](I,\gamma) = [\![\texttt{<regx><anchor>}]\!](I,\gamma)$

8) $[\![\texttt{"/" Regex}(\sigma)\texttt{ "/"}]\!](s) = \textit{true}$ iff regular expression σ matches `String` s.

9) $[\![\texttt{":"<spatialc><regx>}]\!](I,\gamma) = \text{EvalNamed}(\bot, \texttt{<spatialc>}, \texttt{<regx>":"}\epsilon_a, I, \gamma)$

10) $[\![\epsilon_a]\!](I,\gamma) = [(\text{Value}(I,\gamma),\gamma)]$

11) $[\![\texttt{<vdir><quant>}_v\texttt{<hdir><quant>}_h]\!](\gamma,\Gamma) = [\![\texttt{<quant>}_v]\!](\textit{true},\gamma,[\![\texttt{<quant>}_h]\!](\textit{false},\gamma,[\![\texttt{<vdir>}]\!](\gamma,[\![\texttt{<hdir>}]\!](\gamma,\Gamma))))$

12) $[\![\texttt{<name>}]\!] =$ a String.

13) $[\![\mathbb{N}]\!](v,\gamma,\Gamma) = [\gamma' \in \Gamma \mid \text{Distance}(v,\gamma,\gamma') = \mathbb{N}]$

14) $[\![\texttt{*}]\!](v,\gamma,\Gamma) = \Gamma$

15) $[\![\texttt{*?}]\!](v,\gamma,\Gamma) = [\text{argmin}_{\gamma'\in\Gamma}\,\text{Distance}(v,\gamma,\gamma')]$

16) $[\![\texttt{*\#}]\!](v,\gamma,\Gamma) = [\text{argmax}_{\gamma'\in\Gamma}\,\text{Distance}(v,\gamma,\gamma')]$

17) $[\![\epsilon_q]\!](v,\gamma,\Gamma) = [\![1]\!](v,\gamma,\Gamma)$

18) $[\![\epsilon_h]\!](\gamma,\Gamma) = [\gamma' \in \Gamma \mid \text{fst}(\gamma) = \text{fst}(\gamma')]$

19) $[\![\epsilon_v]\!](\gamma,\Gamma) = [\gamma' \in \Gamma \mid \text{snd}(\gamma) = \text{snd}(\gamma')]$

20) $[\![\texttt{"u"}]\!](\gamma,\Gamma) = [\gamma' \in \Gamma \mid \text{snd}(\gamma') \le \text{snd}(\gamma)]$

21) $[\![\texttt{"d"}]\!](\gamma,\Gamma) = [\gamma' \in \Gamma \mid \text{snd}(\gamma') \ge \text{snd}(\gamma)]$

22) $[\![\texttt{"l"}]\!](\gamma,\Gamma) = [\gamma' \in \Gamma \mid \text{fst}(\gamma') \le \text{fst}(\gamma)]$

23) $[\![\texttt{"r"}]\!](\gamma,\Gamma) = [\gamma' \in \Gamma \mid \text{fst}(\gamma') \ge \text{fst}(\gamma)]$

24) $\text{ToTuple}(I,\tau) \equiv \langle \text{Map}(\lambda\phi'.\text{fst}(\phi') : \text{Value}(I,\text{snd}(\phi')), \text{Filter}(\lambda\phi.\text{fst}(\phi) \neq \bot)\tau)\rangle$

25) $\text{Eval}(\texttt{<spatialc>},\texttt{<cellc>},I,\gamma) \equiv \text{EvalNamed}(\text{GetName}(\texttt{<cellc>}),\texttt{<spatialc>},\texttt{<cellc>},I,\gamma)$

26) $\text{GetName}(\texttt{"<" <name> "," <regx> ">" <anchor>}) \equiv \texttt{<name>}$

27) $\text{GetName}(\texttt{<regx><anchor>}) \equiv \bot$

28) $\text{EvalNamed}(\texttt{<name>},\texttt{<spatialc>},\texttt{<cellc>},I,\gamma) \equiv \text{Map}(\lambda\gamma'.\text{Pair}([\![n]\!],\gamma'),[\![s]\!](\gamma,[\gamma'' \in \text{AllCells}(I) | [\![c]\!](I,\gamma'')]))$
$\text{where } n = \texttt{<name>}, s = \texttt{<spatialc>}, \text{ and } c = \texttt{<cellc>}$

29) $\text{Distance}(v,\gamma,\gamma') \equiv |\text{Sel}(v,\gamma) - \text{Sel}(v,\gamma')|$

30) $\text{Sel}(\textit{true},\gamma) \equiv \text{snd}(\gamma)$

31) $\text{Sel}(\textit{false},\gamma) \equiv \text{fst}(\gamma)$

Figure A.2: Formal semantics for the FLARE language. Lines 24-31 are macros. `::` is the list `cons` operator. `[]` is an empty list. Kleene `+` is omitted for space. v is a Boolean value. See Fig. A.1 for other variable types. `<anchor>` is evaluated as if it is an `<cpair>` with the column name $\bot$. "non-captured" columns named $\bot$ are removed by ToTuple.

SYNTH(I,P,N)

1 **for each** column index i
2 $\quad \text{A}_\mathcal{C}[i] = \text{Learn}\mathcal{C}(I,P,i)$
3 **for each** pair of column names i,j such that $i \neq j$
4 $\quad \text{A}_\mathcal{S}[i,j] = \text{Learn}\mathcal{S}(I,P,i,j)$
5 **return** SEARCH$(\emptyset,N,\text{A}_\mathcal{C},\text{A}_\mathcal{S})$

Figure A.3: SYNTH algorithm. FLASHRELATE's top-level synthesis procedure. I is the input spreadsheet, P is the set of positive example tuples, and N is the set of negative example tuples. The procedure precomputes all constraints satisfying P and then calls the search routine.

```
SEARCH(C_P, N, A_C, A_S)
 1  if |C_P| = NUMCOLS
 2       if ⋃_{(c,s)∈C_P} Negate(c) ∪ Negate(s) = N
 3            return C_P
 4       else return FAILURE
 5  else
 6       pairs = {(c, s) | c ∈ A_C[i], s ∈ A_S[i, j]
                  s.t. G' = (V, C_P ∪ {(c, s)}) is loop-free
                  and i, j ∈ [1 ... NUMCOLS]}
 7       pairs' = RANKP(pairs)
 8       k = 0
 9       while k < |pairs'|
10            C'_P = SEARCH(C_P ∪ {pairs'[k]}, N, A_C, A_S)
11            if C'_P ≠ FAILURE
12                 return C'_P
13            k = k + 1
14       return FAILURE
```

Figure A.4: FLASHRELATE's program search procedure. The output is a set of edges guaranteed to be a spanning tree. The routine that inserts child-of operators is omitted for brevity.

```
LEARNC(I, P, i)
1  A = set of predefined constraints
2  A_C = ∅
3  for each constraint α ∈ A
4       if ∀ tuples p ∈ P, [[α]] (I, p[i]) = true
5            A_C = A_C ∪ {Cell(i, α)}
6  return A_C
```

Figure A.5: Learn$\mathcal{C}$ algorithm. Learn$\mathcal{C}$ learns cell constraints from positive examples; `Cell(`i, α`)` is a cell constraint constructor that takes a column name i and a regular expression α. $p \in P$ is a tuple, and when indexed by a column name, yields a cell (x, y).

LEARN$\mathcal{S}(I, P, i, j)$

1 $\mathrm{A}_{\mathcal{S}} = \emptyset$
2 $V =$ LEARNDIRAMOUNT$(I, P, i, j,$ TRUE$)$
3 $H =$ LEARNDIRAMOUNT$(I, P, i, j,$ FALSE$)$
4 **for each** $\{v, h \mid v \in V, h \in H\}$
5 $\quad \mathrm{A}_{\mathcal{S}} = \mathrm{A}_{\mathcal{S}} \cup \{$`Spatial(`$i$`,`$j$`,`$v$`,`$h$`)`$\}$
6 **return** $\mathrm{A}_{\mathcal{S}}$

Figure A.6: Learn$\mathcal{S}$ algorithm. Learn$\mathcal{S}$ learns spatial constraints from positive examples; `Spatial(`i`,`j`,`v`,`h`)` is a spatial constraint constructor that takes two column names i and j, a `<vert>` v, and a `<horiz>` h. LEARNDIRAMOUNT is a function that enumerates geometric descriptors; see Chapter 2.4 "Pruning".

APPENDIX B

NUMBER OF RESAMPLES

For an input vector of length m and a given value from that vector, x, the probability of randomly selecting a value that is not x is $\frac{m-1}{m}$. The probability of selecting m such values is therefore $(\frac{m-1}{m})^m$. As m grows, we obtain the following identity:

$$\lim_{m\to\infty}\big(\frac{m-1}{m}\big)^m = \frac{1}{e}$$

Proof.

$$\begin{aligned}
\lim_{m\to\infty}\big(\frac{m-1}{m}\big)^m &= \lim_{m\to\infty} e^{\ln\big(\frac{m-1}{m}\big)^m} && \Big| x = e^{\ln x} \\
&= \lim_{m\to\infty} e^{m\cdot\ln\big(\frac{m-1}{m}\big)} && \Big| \log b^a = a\cdot\log b \\
&= e^{\lim_{m\to\infty} m\cdot\ln\big(\frac{m-1}{m}\big)} && \Big| \text{If } f \text{ is continuous at } b \text{ and } \lim_{x\to a} g(x) = b, \\
& && \Big| \text{then } \lim_{x\to a} f(g(x)) = f(b) = f(\lim_{x\to a} g(x)). \\
&= e^{-\lim_{m\to\infty}\frac{m}{m-1}} && \Big| \text{L'Hôpital's rule, etc.} \\
&= e^{-\lim_{m\to\infty}\frac{1}{1-\frac{1}{m}}} && \Big| \text{Divide by } m. \\
&= \frac{1}{e} && \Big| \text{Evaluate } m \text{ at } \infty.
\end{aligned}$$

□

BIBLIOGRAPHY

[1] Abraham, Robin, and Erwig, Martin. Header and unit inference for spreadsheets through spatial analyses. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on* (2004), IEEE, pp. 165–172.

[2] Abraham, Robin, and Erwig, Martin. Inferring templates from spreadsheets. In *Proceedings of the 28th international conference on Software engineering* (2006), ACM, pp. 182–191.

[3] Ahmad, Yanif, Antoniu, Tudor, Goldwater, Sharon, and Krishnamurthi, Shriram. A type system for statically detecting spreadsheet errors. In *ASE* (2003), IEEE Computer Society, pp. 174–183.

[4] Ait-Ameur, Y., Bel, G., Boniol, F., Pairault, S., and Wiels, V. Robustness analysis of avionics embedded systems. *SIGPLAN Not. 38*, 7 (June 2003), 123–132.

[5] Antoniu, Tudor, Steckler, Paul A., Krishnamurthi, Shriram, Neuwirth, Erich, and Felleisen, Matthias. Validating the unit correctness of spreadsheet programs. In *Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), ICSE '04, IEEE Computer Society, pp. 439–448.

[6] Ash, Michael, and Pollin, Robert. Supplemental Technical Critique of Reinhart and Rogoff, "Growth in a Time of Debt". Research brief, Political Economy Research Institute, University of Massachusetts Amherst, Apr. 2013.

[7] Asuncion, Hazeline U. Automated data provenance capture in spreadsheets, with case studies. *Future Gener. Comput. Syst. 29*, 8 (Oct. 2013), 2169–2181.

[8] Aurigemma, Salvatore, and Panko, Raymond R. The detection of human spreadsheet errors by humans versus inspection (auditing) software. *CoRR abs/1009.2785* (2010).

[9] Barik, Titus, Lubick, Kevin, Smith, Justin, Slankas, John, and Murphy-Hill, Emerson R. Fuse: A reproducible, extendable, internet-scale corpus of spreadsheets. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015* (2015), pp. 486–489.

[10] Barnett, V., and Lewis, T. Outliers in statistical data. *Wiley Series in Probability and Mathematical Statistics. Applied Probability and Statistics, Chichester: Wiley, 1994, 3rd ed. 1* (1994).

[11] Barowy, Daniel W., Gochev, Dimitar, and Berger, Emery D. Checkcell: Data debugging for spreadsheets. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 507–523.

[12] Barowy, Daniel W, Hart, Sumit Gulwani Ted, and Zorn, Benjamin. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples.

[13] Batty, Michael. Spatial entropy. *Geographical Analysis 6*, 1 (1974), 1–31.

[14] Bessey, Al, Block, Ken, Chelf, Ben, Chou, Andy, Fulton, Bryan, Hallem, Seth, Henri-Gros, Charles, Kamsky, Asya, McPeak, Scott, and Engler, Dawson. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75.

[15] Bolton, Richard J, and Hand, David J. Statistical fraud detection: A review. *Statistical science* (2002), 235–249.

[16] Cafarella, Michael J, Halevy, Alon, and Madhavan, Jayant. Structured data on the web. *CACM 54*, 2 (2011), 72–79.

[17] Carver, Jeffrey, Fisher, II, Marc, and Rothermel, Gregg. An empirical evaluation of a testing and debugging methodology for excel. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* (New York, NY, USA, 2006), ISESE '06, ACM, pp. 278–287.

[18] Chambers, Chris, and Erwig, Martin. Reasoning about spreadsheets with labels and dimensions. *J. Vis. Lang. Comput. 21*, 5 (Dec. 2010), 249–262.

[19] Chang, Chia-Hui, and Lui, Shao-Chen. Iepad: information extraction based on pattern discovery. In *WWW* (2001).

[20] Chaudhuri, Surajit, and Dayal, Umeshwar. An overview of data warehousing and OLAP technology. *SIGMOD Rec. 26*, 1 (Mar. 1997), 65–74.

[21] Chen, Zhe, and Cafarella, Michael. Automatic web spreadsheet data extraction. In *SSW'13* (2013).

[22] Chen, Zhe, and Cafarella, Michael. Integrating spreadsheet data via accurate and low-effort extraction. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2014), KDD '14, ACM, pp. 1126–1135.

[23] Chen, Zhe, Cafarella, Michael, Chen, Jun, Prevo, Daniel, and Zhuang, Junfeng. Senbazuru: a prototype spreadsheet database management system. *PVLDB 6*, 12 (2013), 1202–1205.

[24] Cheung, Shing-Chi, Chen, Wanjun, Liu, Yepang, and Xu, Chang. CUSTODES: Automatic spreadsheet cell clustering and smell detection using strong and weak features. In *Proceedings of ICSE '16* (May 2016). to appear.

[25] Chilimbi, Trishul M., and Ganapathy, Vinod. HeapMD: Identifying heap-based bugs using anomaly detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 219–228.

[26] Cooper, Keith D., and Torczon, Linda. *Engineering a Compiler*. Morgan Kaufmann, 2005.

[27] Crescenzi, Valter, Mecca, Giansalvatore, and Merialdo, Paolo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB* (2001).

[28] Cunha, Jácome, Saraiva, João, and Visser, Joost. From spreadsheets to relational databases and back. In *PEPM 2009* (2009), ACM, pp. 179–188.

[29] Dimitrov, Martin, and Zhou, Huiyang. Anomaly-based bug prediction, isolation, and validation: An automated approach for software debugging. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 61–72.

[30] Dou, Wensheng, Cheung, Shing-Chi, and Wei, Jun. Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 848–858.

[31] Dou, Wensheng, Xu, Liang, Cheung, Shing-Chi, Gao, Chushu, Wei, Jun, and Huang, Tao. Venron: A versioned spreadsheet corpus and related evolution analysis. In *Proceedings of the 38th International Conference on Software Engineering Companion* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 162–171.

[32] Efron, B. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics 7*, 1 (1979), pp. 1–26.

[33] Engels, Gregor, and Erwig, Martin. Classsheets: automatic generation of spreadsheet applications from object-oriented specifications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (2005), ACM, pp. 124–133.

[34] Engler, Dawson, Chen, David Yu, Hallem, Seth, Chou, Andy, and Chelf, Benjamin. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 57–72.

[35] Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., and Xiao, C. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 1 (2007), 35–45.

[36] Erwig, Martin. Software engineering for spreadsheets. *IEEE Softw. 26*, 5 (Sept. 2009), 25–30.

[37] Erwig, Martin, Abraham, Robin, Cooperstein, Irene, and Kollmansberger, Steve. Automatic generation and maintenance of correct spreadsheets. In *ICSE* (New York, NY, USA, 2005), ICSE '05, ACM, pp. 136–145.

[38] Erwig, Martin, and Burnett, Margaret. Adding apples and oranges. In *Practical Aspects of Declarative Languages*. Springer, 2002, pp. 173–191.

[39] Ferrara, Emilio, De Meo, Pasquale, Fiumara, Giacomo, and Baumgartner, Robert. Web data extraction, applications and techniques: a survey. *arXiv preprint arXiv:1207.0246* (2012).

[40] Fisher, Kathleen, and Walker, David. The PADS project: an overview. In *ICDT* (2011).

[41] Fisher, M., Rothermel, G., Creelan, T., and Burnett, M. Scaling a dataflow testing methodology to the multiparadigm world of commercial spreadsheets. In *17th International Symposium on Software Reliability Engineering (ISSRE'06)* (2006), IEEE, pp. 13–22.

[42] Fisher, Marc, and Rothermel, Gregg. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. *SIGSOFT Softw. Eng. Notes* (July 2005).

[43] Fisher, Marc II, and Rothermel, Gregg. The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *1st WEUSE* (2005), pp. 47–51.

[44] Galhardas, Helena, Florescu, Daniela, Shasha, Dennis, and Simon, Eric. Ajax: an extensible data cleaning tool. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2000), SIGMOD '00, ACM, p. 590.

[45] Golab, Lukasz, Karloff, Howard, Korn, Flip, and Srivastava, Divesh. Data auditor: exploring data quality and semantics using pattern tableaux. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 1641–1644.

[46] Gulwani, Sumit. Synthesis from examples: Interaction models and algorithms. In *SYNASC* (2012).

[47] Hamlet, Dick. Continuity in software systems. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2002), ISSTA '02, ACM, pp. 196–200.

[48] Han, J., and Kamber, M. *Data mining: concepts and techniques*. Morgan Kaufmann, 2006.

[49] Hangal, Sudheendra, and Lam, Monica S. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ICSE '02, ACM, pp. 291–301.

[50] Harris, William R., and Gulwani, Sumit. Spreadsheet table transformations from examples. In *PLDI* (2011).

[51] Hellerstein, J.M. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)* (2008).

[52] Hermans, Felienne, Pinzger, Martin, and van Deursen, Arie. Automatically extracting class diagrams from spreadsheets. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings* (2010), pp. 52–75.

[53] Hermans, Felienne, Pinzger, Martin, and van Deursen, Arie. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering 20*, 2 (Apr 2015), 549–575.

[54] Hernández, Mauricio A., and Stolfo, Salvatore J. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1995), SIGMOD '95, ACM, pp. 127–138.

[55] Herndon, Thomas, Ash, Michael, and Pollin, Robert. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. Working Paper Series 322, Political Economy Research Institute, University of Massachusetts Amherst, Apr. 2013.

[56] Hofer, Birgit, Riboira, André, Wotawa, Franz, Abreu, Rui, and Getzner, Elisabeth. On the empirical evaluation of fault localization techniques for spreadsheets. In *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering* (Berlin, Heidelberg, 2013), FASE'13, Springer-Verlag, pp. 68–82.

[57] Hsu, Chun-Nan, and Dung, Ming-Tzung. Generating finite-state transducers for semi-structured data extraction from the web. *Inf. Syst. 23*, 9 (1998).

[58] Igarashi, Takeo, Mackinlay, Jock D, Chang, Bay-Wei, and Zellweger, Polle T. Fluid visualization of spreadsheet structures. In *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on* (1998), IEEE, pp. 118–125.

[59] Jeffery, S.R., Alonso, G., Franklin, M.J., Hong, Wei, and Widom, J. A pipelined framework for online cleaning of sensor data streams. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)* (Apr. 2006), pp. 140–142.

[60] Ko, Andrew J., Abraham, Robin, Beckwith, Laura, Blackwell, Alan, Burnett, Margaret, Erwig, Martin, Scaffidi, Chris, Lawrance, Joseph, Lieberman, Henry, Myers, Brad, Rosson, Mary Beth, Rothermel, Gregg, Shaw, Mary, and Wiedenbeck, Susan. The state of the art in end-user software engineering. *ACM Comput. Surv. 43*, 3 (Apr. 2011), 21:1–21:44.

[61] Kruskal, J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society 7*, 1 (Feb. 1956), 48–50.

[62] Kushmerick, Nicholas, Weld, Daniel S., and Doorenbos, Robert B. Wrapper induction for information extraction. In *IJCAI (1)* (1997).

[63] Le, Vu, and Gulwani, Sumit. FlashExtract: A Framework for Data Extraction by Examples. In *PLDI* (2014), pp. 542–553.

[64] Lieberman, H. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[65] Lohr, Steve. For big-data scientists, janitor work is key hurdle to insights. Online: `https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html`, Last Accessed: 13 Dec 2016.

[66] Lu, Edward, Bodik, Ras, and Hartmann, Bjrn. Quicksilver: Automatic Synthesis of Relational Queries. Tech. Rep. UCB/EECS-2013-68, UC-Berkeley, May 2013.

[67] Luebbers, Dominik, Grimmer, Udo, and Jarke, Matthias. Systematic development of data mining-based data quality tools. In *Proceedings of the 29th International Conference on Very Large Data Bases* (2003), VLDB '03, VLDB Endowment, pp. 548–559.

[68] Microsoft Corporation. Component Object Model (COM). `https://msdn.microsoft.com/en-us/library/windows/desktop/ms680573(v=vs.85).aspx`.

[69] Microsoft Corporation. Channel 9: Going Deep: Craig Symonds and Mohsen Agsen: C++ Renaissance. `https://channel9.msdn.com/Shows/Going+Deep/Craig-Symonds-and-Mohsen-Agsen-C-Renaissance`, Feb. 2011.

[70] Microsoft Corporation. Regular Expression Language - Quick Reference. `https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference`, Mar. 2017.

[71] Monge, Gaspard. Mémoire sur la théorie des déblais et des remblais. *Histoire de l'Académie Royale des Sciences* (1781), 666–704.

[72] Muslea, Ion, Minton, Steven, and Knoblock, Craig A. A hierarchical approach to wrapper induction. In *Agents* (1999).

[73] Nigrini, Mark. *Benford's Law: Applications for forensic accounting, auditing, and fraud detection*, vol. 586. John Wiley & Sons, 2012.

[74] Oro, Ermelinda, and Ruffolo, Massimo. Sila: a spatial instance learning approach for deep webpages. In *CIKM 2011* (2011), ACM, pp. 2329–2332.

[75] Oro, Ermelinda, Ruffolo, Massimo, and Staab, Steffen. Sxpath: extending xpath towards spatial querying on web documents. *PVLDB 4*, 2 (2010), 129–140.

[76] Panko, Ray. What we don't know about spreadsheet errors today: The facts, why we dont believe them, and what we need to do. In *The European Spreadsheet Risks Interest Group 16th Annual Conference* (2015), EuSpRiG 2015, EuSpRiG.

[77] Panko, Raymond R. What we know about spreadsheet errors. *Journal of End User Computing 10* (1998), 15–21.

[78] Panko, Raymond R., and Halverson, Richard P. n Experiment in Collaborative Spreadsheet Development. *Journal of the Association for Information Systems 2*, 4 (2001).

[79] Quinlan, J. R. Induction of decision trees. *MACH. LEARN 1* (1986), 81–106.

[80] Rahm, Erhard, and Do, Hong Hai. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull. 23*, 4 (2000), 3–13.

[81] Raman, Vijayshankar, and Hellerstein, Joseph M. Potter's wheel: An interactive data cleaning system. In *Proceedings of the 27th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2001), VLDB '01, Morgan Kaufmann Publishers Inc., pp. 381–390.

[82] Raz, Orna, Koopman, Philip, and Shaw, Mary. Semantic anomaly detection in online data sources. In *ICSE* (New York, NY, USA, 2002), ICSE '02, ACM, pp. 302–312.

[83] Reinhart, Carmen M., and Rogoff, Kenneth S. Growth in a time of debt. Working Paper 15639, National Bureau of Economic Research, January 2010.

[84] Reinhart, Carmen M, and Rogoff, Kenneth S. Growth in a time of debt. *The American Economic Review 100*, 2 (2010), 573–78.

[85] Rothermel, G., Burnett, M., Li, L., Dupuis, C., and Sheretov, A. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology (TOSEM) 10*, 1 (2001), 110–147.

[86] Rothermel, G., Li, L., DuPuis, C., and Burnett, M. What you see is what you test: A methodology for testing form-based visual programs. In *ICSE 1998* (1998), IEEE, pp. 198–207.

[87] Shannon, C. E. A mathematical theory of communication. *Bell system technical journal 27* (1948).

[88] Solar-Lezama, Armando, Tancau, Liviu, Bodik, Rastislav, Seshia, Sanjit, and Saraswat, Vijay. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 404–415.

[89] Verborgh, Ruben, and De Wilde, Max. *Using OpenRefine*. Packt Publishing, Sept. 2013.

[90] Wegner, Peter. A technique for counting ones in a binary computer. *Commun. ACM 3*, 5 (May 1960), 322–.

[91] Welsh, D. J. A., and Powell, M. B. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal 10*, 1 (1967), 85–86.

[92] Wikipedia. Microsoft Excel — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=Microsoft_Excel&oldid=714358324`, 2016. [Online; accessed 12-April-2016].

[93] Xie, Yichen, and Engler, Dawson. Using redundancies to find errors. In *IEEE Transactions on Software Engineering* (2002), pp. 51–60.

[94] Xiong, H., Pandey, Gaurav, Steinbach, M., and Kumar, Vipin. Enhancing data analysis with noise removal. *IEEE Transactions on Knowledge and Data Engineering 18*, 3 (Mar. 2006), 304–319.

[95] Zeller, Andreas. *Why Programs Fail: A Guide to Systematic Debugging*, 2 ed. Morgan Kaufmann, June 2009.