Data Debugging

Daniel W. Barowy Dimitar Gochev Emery D. Berger

School of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{dbarowy,gochev,emery}@cs.umass.edu

Abstract

Testing and static analysis can help root out bugs in programs, but not in data. This paper introduces *data debugging*, an approach that combines program analysis and statistical analysis to find potential data errors. Since it is impossible to know *a priori* whether data are erroneous or not, data debugging locates data that has an unusual impact on the computation. Such data is either very important, or wrong. Data debugging is especially useful in the context of data-intensive programming environments that intertwine data with programs in the form of queries or formulas. We present the first data debugging tool, CHECKCELL, an add-in for Microsoft Excel. CHECKCELL colors cells red when they have an unusually high impact on the spreadsheet's computations. We show that CHECKCELL is both analytically and empirically fast and effective; in a case study, it automatically identifies a key flaw in the infamous Reinhart and Rogoff spreadsheet.

1. Introduction

In many computational tasks, correctness is a primary concern. Most work in the programming language community has focused on ways to discover whether the program performing the computation is correct. Techniques to reduce program errors range from testing and runtime assertions, to dynamic and static analysis tools that can discover a wide range of bugs. These tools and approaches enable programmers to find errors and reduce their impact, contributing to improving overall code quality.

However, a program is just one part of a computation. Existing tools ignore the correctness of program *inputs*. If the input contains errors, the result of the computation is not likely to be correct. Unlike programs, data cannot be easily tested or analyzed for correctness.

Input data errors can arise in a variety of ways [24]:

- **Data entry errors**, including typographical errors and transcription errors from illegible text.
- **Measurement errors**, when the data source itself, such as a disk or a sensor, is faulty or corrupted (unintentionally or not).
- **Data integration errors**, where inconsistencies arise due to the mixing of different data, including unit of measurement mismatches.

While data errors pose a threat to the correctness of any computation, they are especially problematic in data-intensive programming environments like databases, spreadsheets, and certain scientific computations. In these settings, data correctness can be as important as program correctness ("garbage in, garbage out"). The results produced by the computations—queries, formulas, charts, and other analyses—may be rendered invalid by data errors. These errors can be costly: errors in spreadsheet data have led to losses of millions of dollars [39, 40].

By contrast with the proliferation of tools at a programmer's disposal to find program errors, few tools exist to help find data errors. Part of the problem is that it can be difficult to decide whether any given data element is an error or not. For example, the number `1234` might be correct, or the correct value might be `12.34`. Typographical errors can change data items by orders of magnitude. Unfortunately, finding this kind of mistake via manual data auditing is onerous, unscalable, and error-prone.

Existing approaches to finding data errors include *data cleaning* and *statistical outlier detection*. Data cleaning primarily copes with errors via cross-validation with ground truth data, which may not be present. Statistical outlier detection typically reports data as outliers based on their relationship to a given distribution (e.g., Gaussian). Automatic identification of data distributions is error-prone and can give rise to excessive false positives.

This paper presents **data debugging**, an approach for locating 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 data that has an unusual impact on the computation*. Intuitively, data that has an inordinate impact on the final result is either very important, or it is wrong. By contrast, wrong data whose presence has no particularly unusual effect on the final result does not merit special attention. Data debugging combines data dependence analysis and statistical analysis to find and highlight data proportional to the severity of its impact on the results of a computation.

Data debugging works by first building a data dependence graph of the computations. It then measures data impact by replacing data items with data chosen from the same group (e.g., a range in a spreadsheet formula) and observing the resulting changes in computations that depend on that data. This nonparametric approach allows data debugging to find errors in both numeric and non-numeric data, without any requirement that data follow any particular statistical distribution.

By calling attention to data with unusual impact, data debugging can provide insights into both the data and the computation and reveal errors. We believe data debugging is broadly applicable, though it is especially well-suited for data-intensive programming environments that intertwine data and programs (e.g., with queries and formulas).

This paper presents the first data debugging tool in the form of CHECKCELL, an add-in for Microsoft Excel and Google Spreadsheets. Spreadsheets are one of the most widely-used programming environments, and this domain has recently attracted renewed academic attention [20, 23, 41]. In addition, spreadsheet errors are a well known risk and have led to significant monetary losses in the past, making them an excellent first target for data debugging.

CHECKCELL highlights all inputs whose presence causes function outputs to be dramatically different than the function output were those outputs excluded. The brightness of the highlighting is proportional both to the severity of the output's unusualness and to the importance of the function in the computation 3. CHECKCELL is empirically and analytically efficient and effective, as we show in Sections 4 and 5. The current prototype is untuned but analysis time is generally low, taking a median of 13 seconds to run on most of the spreadsheets we examine. By employing human workers via Amazon's Mechanical Turk crowdsourcing platform to generate errors, we show that CHECKCELL is effective at finding actual data entry errors in a random selection of spreadsheets from the EUSES corpus. We also apply CHECKCELL to a real-world spreadsheet: CHECKCELL automatically identifies a key flaw in the now-infamous Reinhart-Rogoff spreadsheet [26].

Contributions

The contributions of this paper are the following:

1. We introduce *data debugging*, an approach aimed at identifying data that has an unusual impact on the final computation, indicating that the data is either extremely important or wrong.
2. We describe novel algorithms to implement data debugging that combine program analysis and nonparametric statistical analysis to identify potential data errors.
3. We present a prototype data debugging tool for spreadsheets, CHECKCELL, and demonstrate its effectiveness at finding errors and identifying highly important data.

Outline

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the algorithms that data debugging employs. Section 4 derives analytical results that demonstrate data debugging's runtime efficiency and effectiveness. Section 5 presents an empirical evaluation of data debugging in the form of CHECKCELL, measuring its runtime performance and its effectiveness at finding errors. Section 6 describes directions for future work, and Section 7 concludes.

2. Related Work

Data Cleaning

Most past work on locating or removing errors in data has focused on *data cleaning* or *scrubbing* in database systems [22, 32]. Standard approaches include statistical outlier analysis for removing noisy data [42], interpolation to fill in missing data (e.g., with averages), and using cross-correlation with other data sources to correct or locate errors [25].

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 [18]. Data Auditor applies rules and target relations entered by a programmer [19]. A similar domain-specific approach has been employed for data streams to smooth data temporally and isolate it spatially [29]. Potter's Wheel, by Raman and Hellerstein, is an interactive tool that lets users visualize and apply data cleansing transformations [33].

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 [31]. Raz et al. describe an approach aimed at arbitrary software that uses Daikon [13] to infer invariants about numerical input data and then report discrepancies as "semantic anomalies" [34]. 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.

Spreadsheet Errors

Spreadsheets have been one of the most prominent computer applications since their creation in 1979. The most widely used spreadsheet application today is Microsoft Excel. Excel includes rudimentary error detection including errors in formula entry like division by zero, a reference to a non-existient formula or cell, invalid numerical arguments, or accidental mixing of text and numbers. Excel also checks for inconsistency with adjacent formulas and other structural errors, which it highlights with a "squiggly" underline. In addition, Excel provides a formula auditor, which lets users view dependencies flowing into and out of particular formulas.

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, 9, 14, 15, 30]. For example, XeLda checks if 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 [8, 17, 27, 30, 37, 38].

This work is complementary and orthogonal to CHECKCELL, which works with standard, unannotated spreadsheets and focuses on unusual interactions of data with formulas.

Statistical Outlier Analysis

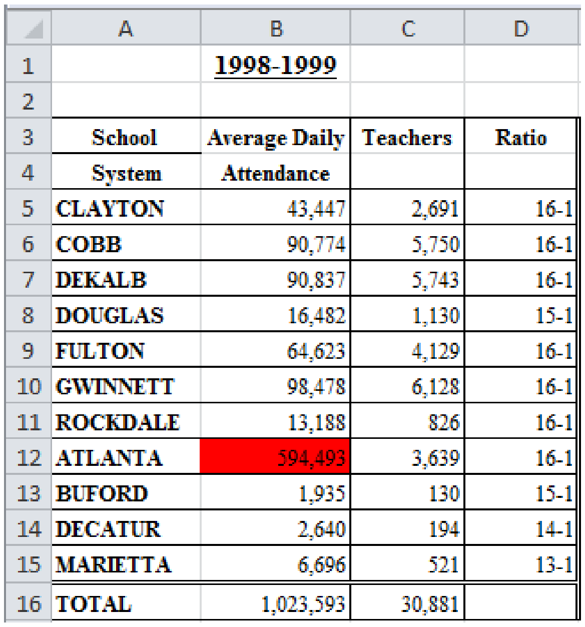
Techniques to locate outliers date to the earliest days of statistics, when they were developed to make nautical measurements more robust. Widely-used approaches include Chauvenet's criterion, Peirce's criterion, and Grubb's test for outliers [7]. All of these techniques are *parametric*: they require that the data belong to a known distribution, generally the Gaussian (normal). Unfortunately, input data does not necessarily fit 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.

Sensitivity Analysis and Uncertainty Quantification

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 [2, 10, 21].

Uncertainty quantification draws a relationship between the uncertainty of an input parameter and the uncertainty in the output. Unlike sensitivity analysis, which in the case of OAT can be used as a "black-box" technique, uncertainty quantification requires the analyst to know the functional composition of the model being analyzed.

Data debugging differs from sensitivity analysis and uncertainty quantification in several important respects. First, data debugging is a fully-automated black-box technique that requires no knowledge of a program's structure. 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 empirical input distribution via a nonparametric statistical approach. Finally, the uncertainty of inputs and outputs is irrelevant to CHECKCELL's analysis. CHECKCELL instead seeks to find specific data elements that have an extraordinary effect on



	A	B	C	D
1		1998-1999		
2				
3	School	Average Daily	Teachers	Ratio
4	System	Attendance		
5	CLAYTON	43,447	2,691	16-1
6	COBB	90,774	5,750	16-1
7	DEKALB	90,837	5,743	16-1
8	DOUGLAS	16,482	1,130	15-1
9	FULTON	64,623	4,129	16-1
10	GWINNETT	98,478	6,128	16-1
11	ROCKDALE	13,188	826	16-1
12	ATLANTA	594,493	3,639	16-1
13	BUFORD	1,935	130	15-1
14	DECATUR	2,640	194	14-1
15	MARIETTA	6,696	521	13-1
16	TOTAL	1,023,593	30,881	

Figure 1. A sample spreadsheet from the EUSES corpus with a real typographical error, highlighted in red (see Section 3).

program outputs. In essence, sensitivity analysis and uncertainty quantification are aimed at analyzing the model, while data debugging is a technique for analyzing the data itself.

3. Data Debugging: Algorithms

This section describes in detail the algorithms that data debugging employs. Section 4 includes formal analysis of various aspects of the data debugging algorithms described here, including asymptotic performance and statistical effectiveness.

Throughout this section, we illustrate these algorithms with a running example of a budget shown in Figure 1, which is an excerpt from a real spreadsheet found in the EUSES corpus. An error was introduced into the spreadsheet by asking users on Mechanical Turk to enter the data into a spreadsheet based web form. The experimental setup is described in more detail in Section 5. This spreadsheet tracks student-teacher ratios for a number of metropolitan public school systems. The user accidentally typed an additional "4", causing an order-of-magnitude error.

3.1 Dependence Analysis

CHECKCELL's statistical analysis is guided by the structure of the program present in a worksheet. CHECKCELL's first step is to identify the inputs and outputs of those computations. CHECKCELL scans the open Excel workbook and collects all formula strings. Formulas are parsed using an Excel grammar expressed with the FParsec parser combinator library. CHECKCELL uses the Excel formula's syntax tree to extract references to input ranges and other formulas. CHECKCELL resolves references to local, cross-worksheet, and cross-workbook cells.

Spreadsheet programs are always strictly tree-shaped, with inputs at the leaves, and with an output at the root. Both the root and all the intermediate nodes of the tree are formulas. The purpose of CHECKCELL is to determine the effect of a particular input on the final output of the program tree, so while intermediate nodes are an important part of a spreadsheet's calculation, their values are not a part of our analysis. There can be many such trees in a spreadsheet, and this forest of functions may even share input leaves. CHECKCELL uses techniques similar to past work to identify dependencies in spreadsheets [17]. For our purposes, charts are simply considered to be function outputs.

CHECKCELL's statistical analysis depends on the ability of the analysis to replace input values with other representative values. When function has only a scalar argument, namely a single cell or a constant, CHECKCELL does not have enough information to reliably generate other representative values. Therefore, CHECKCELL limits its analysis to vector inputs.

3.2 Impact Analysis

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

Removing an input value requires replacing it with another representative value. Since CHECKCELL 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. CHECKCELL thus operates under the additional assumption that values in vector inputs are exchangeable.

Function Classes

CHECKCELL assumes that inputs to functions that it examines are homogeneous, i.e., that a vector input can be considered an order-independent, i.i.d. random samples, drawn from some unknown distribution. Our analysis of frequently-used vector functions shows that the most widely-used functions in Excel are in fact order-independent. Conveniently, this assumption allows us to draw from a large corpus of statistical outlier detection methods.

CHECKCELL does not directly perturb the inputs to vector functions that do not satisfy the homogeneity requirement. Of the 5,606 spreadsheets in the EUSES spreadsheet corpus [16], 4,038 contain spreadsheets for a total of 730,765 formulas. Our comprehensive analysis of these spreadsheets showed that order-dependent vector functions like HLOOKUP, INDEX, VLOOKUP, and OFFSET are largely dominated by order-independent vector functions, particularly SUM. Thus CHECKCELL is useful for a very large number of existing spreadsheets. Figure 2 shows the relative frequency of the ten most common vector functions in the EUSES corpus.

Non-Parametric Methods: the Bootstrap

Standard approaches to outlier rejection generally depend on the shape of the distribution. These so-called *parametric* methods require data analysts to parameterize their hypothesis tests with a known parametric form. The normal distribution is most often assumed for outlier rejection. This assumption is justified primarily when a distribution is known to be the result of a summing or averaging of values, since these values will converge in the limit to the normal distribution according to the Central Limit Theorem. Given that CHECKCELL needs to perform statistical tests on *any* function and over unknown data distributions, parametric methods are inappropriate.

Instead, CHECKCELL's input analysis incorporates an adaptation of Efron's bootstrap procedure, a *non-parametric* (distribution-free) statistical method [12]. The bootstrap is a procedure for estimating the distribution of a function output, referred to as the *test statistic*, given only an approximation of population of interest (typically a sample). This distribution allows one to measure the variability of the test statistic, allowing for reliable inference even when the following conditions hold:

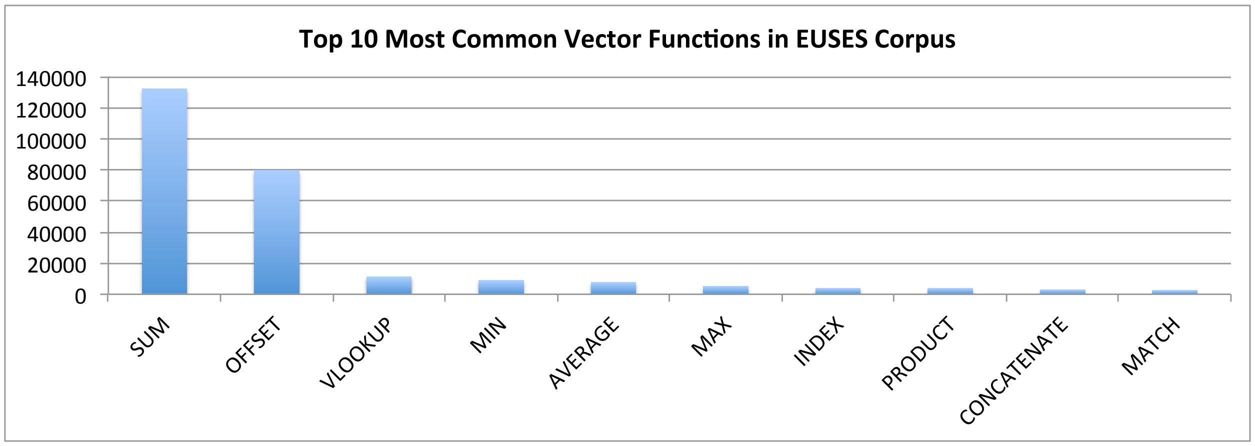


Figure 2. A frequency count of the 10 most common vector functions in the EUSES spreadsheet corpus. The SUM, MIN, AVERAGE, MAX, PRODUCT, MATCH functions are order-independent while OFFSET, VLOOKUP, INDEX, and CONCATENATE are not.

- The approximated distribution is small, *i.e.*, under 30 elements, or
- The distribution is either difficult to compute or is completely unknown.

In particular, CHECKCELL uses an adaptation of Efron's *basic bootstrap* procedure. The procedure works as follows:

1. Draw a random sample, $X_i = (X_0, \ldots, X_m)$, with replacement, from the input vector of interest. This new vector is referred to as a *resample*. Note that m must be the size of the original sample.
2. Compute the test statistic, $\hat{\theta}_i(X_i)$, which is a point estimator for the true test statistic of the population, $\theta(X)$.
3. Repeat this process n times. In the statistical literature, the number of bootstraps typically is between 1000 and 2500; CHECKCELL uses $n = 1000 \cdot e$, which is approximately 2800 (see Section 4.1).

The resulting distribution $\hat{\theta} = (\hat{\theta}_i, \ldots, \hat{\theta_{n-1}})$ gives an approximation of θ, the true value of the test statistic for the population. This distribution can now 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.

Hypothesis test. In order to determine whether an input, x, is likely to be an error, CHECKCELL 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} (a point value), is highly unusual when compared to the $\hat{\theta_e}$, the input x is either a very important input or a likely error. CHECKCELL performs two variants of the hypothesis test, depending on whether the output of the function of interest is numeric or string-valued.

Numeric function outputs. For numeric outputs, the bootstrap distribution is sorted in ascending order, and the quantile function is applied to determine the confidence bound of interest. CHECKCELL uses $\alpha = 0.05$, which is a standard confidence bound in the statistical literature, corresponding to a 95% confidence interval. The original function output is compared with the distribution $\hat{\theta_e}$. If θ_{orig} falls to the left of the 2.5th quantile or to the right of the 97.5th quantile, we reject the null hypothesis and declare x an outlier.

String-valued function outputs. 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 $\sum_{0 \le i \le k-1} p_i = 1$. CHECKCELL calculates p_i from the observed frequency of category i from $\hat{\theta_e}$. The null hypothesis is then rejected if the probability of observing the original function output, θ_{orig}, is less than α. The accuracy of the multinomial hypothesis-testing procedure depends on the number of bootstraps, n, since if $n \ll k$ then $\hat{\theta_e}$ is guaranteed to be sparse and incorrect inferences may be drawn. In principle, n can be adjusted such that we are unlikely to observe a $p_i = 0$ when the true value of $p_i = 0 + \epsilon$. In practice, $n = 1000$ is more than enough for most multinomials encountered.

3.3 Impact Scoring

Finally, all inputs that failed at least one hypothesis test are highlighted and presented to the user. Brightly-colored outputs indicate likely severe outliers while dimly-colored outputs indicate less severe outliers. Input cells that failed no hypothesis test retain their original color (typically black text on a white background). Inputs that do not participate in any computations have no chance of being flagged as potential errors.

CHECKCELL cannot know *a priori* which function outputs are the most important to the end-users. However, inputs which have large effects on large-scale computations are arguably more important to find than inputs that have large effects on small-scale computations. Thus, CHECKCELL weights the *impact score* of a suspected input error for a particular function output is by the size of the computation tree (the number of input leaves) for that function. The *total impact* of the error is thus defined as $\sum_{s_{i,f} \in S} w_f$ where $s_{i,f}$ is the impact score for input i and function f and w_f is the weight of function f. The brightness of the highlighting is $(s_{i,f} - s_{min})/(s_{max} - s_{min})$ where 0 is no highlighting and 1 is the brightest highlight.

3.4 Optimizations

CHECKCELL's runtime is $O(i \cdot n)$, or linear in the number of recalculations required (see Section 4), where i is the number of input vectors and n is the number of bootstraps required. Our system uses a configurable default of $n = 1000 \cdot e$ (see Section 4.1).

As n grows larger than m, the length of an input vector, the probability that a given resample will again appear during the bootstrapping procedure increases substantially. CHECKCELL makes use of this fact to save on recalculation cost by caching the output of functions whose input values have been previously calculated.

CHECKCELL calculates a fingerprint for each resample that lets it identify duplicate resamples. Since the inputs to vector functions are order-invariant, CHECKCELL only needs to track the number of appearances of a particular input value in a resample. The fingerprint is a vector of counters, one for each index in the input. CHECKCELL keeps a dictionary of previously-calculated values of $\hat{\theta}_i$, where the key is the aforementioned fingerprint.

For example, given the input vector $(1, 2, 3, 4)$, one possible resample, $X = (X_0, X_1, X_2, X_3)$, is $(1, 4, 4, 3)$. The fingerprint counter would then be $c_0 = 1, c_1 = 0, c_2 = 1, c_3 = 2$. Section 4.2 analyzes the efficiency of this mechanism.

4. Data Debugging: 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 significantly mitigating this cost is also discussed.

4.1 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 values is therefore $(\frac{m-1}{m})^m$. As m grows, we obtain the following identity:

Lemma 4.1. $\lim_{m\to\infty} (\dfrac{m-1}{m})^m = \dfrac{1}{e}$

Statistical literature suggests that the number of bootstraps be at least 1000 when the computational cost is tolerable. For efficiency reasons, 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 $n = 1000 \cdot e$. Lemma 4.1 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, CHECKCELL requires $O(i \cdot n)$ time to analyze a spreadsheet. In practice, the caching feature described in Section 3.4 makes observing even this modest linear cost highly unlikely.

4.2 Efficiency of Caching

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. Input vectors are resampled uniformly randomly, thus 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.

For example, suppose we have the following vector: ABC. While there are 3^3 possible ways to resample from this vector, a large number of those combinations are not unique when we ignore the ordering of the elements. The complete set of *distinct* order-independent combinations are: AAA, AAB, AAC, ABB, ACC, ABC, BBB, BBC, BCC, CCC. $\binom{2\cdot 3-1}{3} = 10$.

5. Evaluation

We evaluate CHECKCELL across three dimensions: its ability to reduce program error, its ability to reduce end-user effort in fixing errors, and its execution time. We also report on a case study of using CHECKCELL to examine a now-infamous spreadsheet due to Reinhart and Rogoff.

Our evaluation is designed to answer the following questions:

1. Does CHECKCELL identify important errors in spreadsheets?
2. Does using CHECKCELL reduce user effort to identify and correct errors?
3. Is CHECKCELL efficient?

5.1 Error Reduction and User Effort

We evaluated CHECKCELL on a selection of the EUSES corpus by injecting errors according to an empirically-trained model, described later in this section.
Quantifying the importance of errors and user effort requires some care. We derive metrics for these in the following.

Quantifying Error

We consider the "correct" (original) spreadsheet to be a vector S of strings; note that we assume that the spreadsheet prior to error injection is correct. CHECKCELL may identify actual errors in the EUSES spreadsheets, but because we do not know the ground truth, we conservatively treat such reports as false positives.

We then proceed to inject errors in the spreadsheet. We refer to a spreadsheet with n errors injected as spreadsheet S_e. Using CHECKCELL leads to a series of k corrections which form the sequence of corrections $c_1 \ldots c_k$. Note that k is less than or equal to n, since it is possible for CHECKCELL not to identify all errors.

We apply the corrections in sequence, $c_1 \ldots c_k$, producing a partially-corrected version of the fault-injected spreadsheet S_e, namely the spreadsheet $S_{p,k}$. Spreadsheet $S_{p,0}$ is the spreadsheet with no corrections applied (S_e). Spreadsheet $S_{p,n}$ is the spreadsheet with all n corrections applied (S, when $k = n$).

Because spreadsheets contain both numeric and non-numeric data, we treat them separately and then combine their terms into a total error metric.

Let f be a real-valued function over spreadsheet inputs. Then the *numerical error* of f is:

$$\text{err}_{\mathbb{R}}(f, k) = |f(S_{p,k}) - f(S)|$$

Note that it is possible for a sequence of corrections to temporarily increase the numeric error (i.e., $\text{err}_{\mathbb{R}}(f, k) > \text{err}_{\mathbb{R}}(f, k + 1)$), because the effect of multiple errors may combine to reduce total error. Thus, we normalize numerical errors by the most extreme error observed. However, after correcting all n errors, the numerical error is guaranteed to be 0.

The *normalized numerical error* of f is thus:

$$\text{nerr}_{\mathbb{R}}(f, k) = \frac{\text{err}_{\mathbb{R}}(f, k)}{\underset{i\in 0..n}{\text{argmax}}\ \text{err}_{\mathbb{R}}(f, i)}$$

We treat non-numerical ("categorical") errors by using an indicator function which is 1 if it differs in value and 0 otherwise: let g be a *categorical function*. Then the *categorical error* of g is:

$$\text{err}_C(g, k) = \begin{cases} 1 & \text{if } g(S_{p,k}) = g(S) \\ 0 & \text{otherwise.} \end{cases}$$

We then compute the *total error* in a spreadsheet as follows. Let the set of all numeric functions defined in a spreadsheet be F and the set of all categorical functions defined in a spreadsheet be G. Then the total error after k corrections of the spreadsheet is:

$$\text{err}_{tot}(k) = \sum_{f\in F} \text{nerr}_{\mathbb{R}}(f, k) + \sum_{g\in H} \text{err}_C(g, k)$$

Finally, since CHECKCELL is not guaranteed to remove all errors, we define the *remaining error* to be:

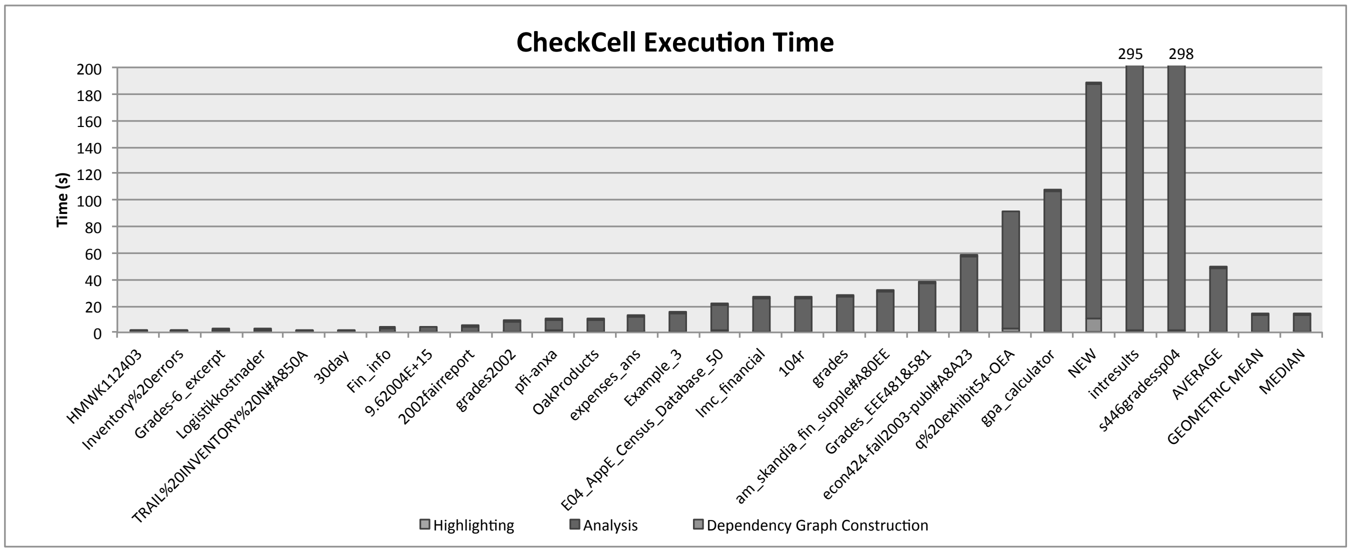


Figure 3. CHECKCELL execution time. For most of the spreadsheets, CHECKCELL completes its analysis in under 40 seconds; for all but two, it completes in under five minutes (see Section 5.2).

$$\text{err}_{rem} = \frac{\text{err}_{tot}(n)}{\text{err}_{tot}(0)}$$

This number expresses the ratio of cells remaining to be fixed. For example, a remaining error of 0.5 means that 50% of the total error remains from the fault-injected spreadsheet. Note that if $k = n$ (we fix all of the errors), then err_{rem} is guaranteed to be 0.

In our evaluation, we report false positives, false negatives, and true positives. A false positive is when CHECKCELL flags a cell as erroneous that is actually correct (as stated earlier, we treat all reports of cells that were not injected with errors as false positives). A false negative is when CHECKCELL fails to flag an injected error as erroneous. A true positive is when CHECKCELL correctly identifies an cell with an injected error.

Quantifying User Effort

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

Experimental Methodology

We claim that CHECKCELL is effective at finding *important data bugs*. However, CHECKCELL specifically looks for inputs that have an unusual impact on the computation, which is not exactly the same thing. To validate our claim that CHECKCELL finds data bugs, we randomly generate inputs that produce a high total error in the computation. In our tests, we inject inputs with a 5% error rate. This rate is consistent with our empirical observations of workers on Mechanical Turk (see below).

We evaluate CHECKCELL by injecting these high-error generated values into a random sample of 25 spreadsheets from the EUSES corpus. We excluded 4 of the 25 benchmarks because they contain a trivial number of input cells (under 10), and so manual inspection of these spreadsheets would be trivial.

We then simulate a user following CHECKCELL's prompts to inspect cells. If the simulated user would inspect a cell that contains a real error, we mark the cell as a true positive and correct the error using the value from the original spreadsheet. If the simulated user inspects a cell that is not an error, we mark the cell as a false positive. After the tool has identified all of the errors at the significance level indicated by the user (we use a default of 95%), all remaining errors are considered to be false negatives. For each error-injected spreadsheet, we compute the *remaining error* and *relative user effort* at the end of the procedure.

If CHECKCELL is successful at removing these generated errors with little user effort, then the tool is effective at removing errors. Since CHECKCELL is the first fully-automated tool for finding data errors, the baseline for our analysis is the requirement to manually inspect every formula input cell.

Error Generator

In order to inject errors that are representative of the kind of errors that people actually make, we built and trained a classifier by recruiting workers on Amazon's Mechanical Turk to perform data entry tasks. The classifier is designed to spot two kinds of errors: (1) character transpositions and (2) other typographical errors.

Our input data came from two sources: we randomly sampled formula inputs from 500 spreadsheets in the EUSES corpus (corresponding to 69,112 input strings), and we randomly generated 100,000 additional strings. The additional strings were created to ensure that users were exposed to a wide range of strings, reducing the sparsity in our model. To make it impossible for users to simply cut and paste these strings back into the input field, we rendered strings as images and had 946 workers re-enter the text shown in the image. Workers correctly re-entered 97.14% strings from the first data set and 93.24% from the second data set for a total accuracy of 94.74% (an error rate of 5.26%).

Results: Error Detection and Effort Reduction

CHECKCELL is effective in automatically finding injected errors. In 42.8% of the cases (9 of 21 cases), CHECKCELL was able to guide simulated users to reduce overall error by roughly 50%, while examining an average of just 4.5 cells (8.3%). A user who did not have CHECKCELL would need to examine 100% of the cells in the spreadsheet–an average of 54 cells (see Figure 4). On average, CHECKCELL had one true positive, 3.5 false positives, and 4.8 false negatives. The fact that CHECKCELL is able to reduce

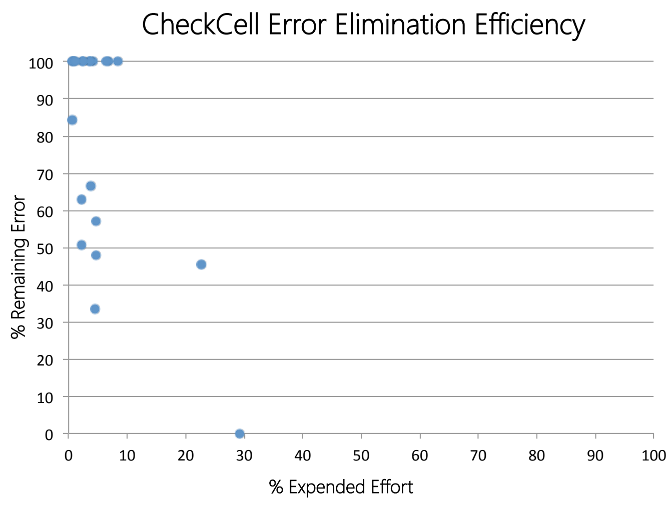


Figure 4. CHECKCELL can efficiently guide users toward finding the most important errors. Note that the ideal tool would place errors in the bottom left of the chart, indicating that important errors are removed with little effort.

error significantly after only a single correction on average strongly supports our claim that CHECKCELL finds the most important errors in a spreadsheet.

5.2 Execution Time

Setup. Our experimental platform is a 2011 Mac Pro equipped 16GB of RAM and a quad-core Intel Xeon processor running at 3.2GHz. The operating system is Windows 7 Professional (SP1), which executes in Parallels. CHECKCELL was compiled using Microsoft Visual Studio 2010, and runs as an add-in in Microsoft Excel 2010. We have also implemented CHECKCELL for Google Spreadsheets, but only report results for the Excel version.

To measure the runtime of CHECKCELL, we ran it on a selection of 30 spreadsheets drawn randomly from the EUSES corpus [16], excluding those that did not contain formulas. CHECKCELL was able to analyze 26 out of the 30 spreadsheets; the 4 it could not analyze contained no vector-valued functions.

Table 1 includes characteristics of these spreadsheets, ordered by the number of formulas each contains. We include two columns that count the number of cells in different ways. *Cells (raw)* indicates the total number of cells that participate in any computation. *Cells (weighted)* indicates the total number of cells *inside ranges*, weighted by the number of times each cell is used in a computation. For example, a cell that is in a range involved in two computations is counted twice. Because the weighted cell count only includes ranges, it is possible for it to be lower than the raw number of cells.

Figure 3 reports the performance of data debugging across our spreadsheet suite, ordered by the weighted number of cells. Table 1 includes the full data.

For 18 of the 26 benchmarks, CHECKCELL takes 30 seconds or less to complete. Its runtime is less than two minutes for all but three of the spreadsheets: `intresults.xls`, `NEW.xls`, and `s446gradessp04xls`, which take 295 seconds, 187 seconds, and 298 seconds, respectively. The average runtime over all spreadsheets is 49 seconds; without the three outliers, it is 21 seconds. As our analysis in Section 4 predicts, the time cost of CHECKCELL is largely dominated to the cost of the impact analysis, which is in turn dependent on the number of inputs.

The spreadsheets that require the most execution time have by far, the largest number of formulas (1,066 and 2,626), and the latter also has the largest number of weighted cells (2,403). Their relatively high execution time is attributable to the fact that the cost of impact analysis increases as the number of formulas increases, since the Excel recalculation engine must do more work per item tested. The `intresults` spreadsheet also has an extremely highly-connected clique in its dependence graph, which leads to both higher time for dependence analysis and increases the cost of recalculations during impact analysis.

Summary: For nearly every spreadsheet examined, CHECKCELL's runtime is under two minutes; we believe this overhead is acceptable for an error detection tool.

5.3 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 [35, 36]. 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 [26].

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 [26]. 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 indicated 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 [5]:

> 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.

6. Future Work

In future work, we plan to explore applying data debugging to other data-intensive domains, including Hadoop/MapReduce tasks [4, 11], scientific computing environments like R [28], and database management systems, especially those with support for "what-if" queries [6].

We expect all of these domains will require some tailoring of the existing algorithms to their particular context. For databases, we plan to treat as computations both stored procedures and cached queries. While it is straightforward to apply data debugging to databases when queries have no side effects, handling queries that do modify the database will take some care in order to avoid an excessive performance penalty due to copying.

A similar performance concern arises with Hadoop, where the key computation is the relatively costly reduction step. Data debugging will also likely need to take into account features of the R language in order to work effectively in that context. Finally, we are interested in exploring the effectiveness of data debugging in conventional programming language settings.

While CHECKCELL's speed is reasonable in most cases, we are interested in further optimizing it. We are especially interested in developing a version that incrementally updates its impacts on-the-fly. This version would run in the background and detect data with unusual impacts as they are entered, much like modern text entry underlines misspelled words. We believe that having automatic detection of possible data errors on all the time could greatly reduce the risk of data errors.

7. Conclusion

This paper 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. CHECKCELL is available for download at `https://checkcell.org`.

Acknowledgments

Thanks to Alexandru Toader, our Google Summer of Code student who ported CHECKCELL to Google Spreadsheets. We gratefully acknowledge Microsoft Research for their support and Ben Zorn for the stimulating conversation that led to this paper. Thanks also to Charlie Curtsinger for the many valuable discussions during the evolution of this project.

References

[1] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A type system for statically detecting spreadsheet errors. In *ASE*, pages 174–183. IEEE Computer Society, 2003.

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

[3] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 439–448, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Apache Foundation. Welcome to Apache Hadoop. `http://hadoop.apache.org/`, Nov. 2012.

[5] M. Ash and R. Pollin. 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.

[6] A. Balmin, T. Papadimitriou, and Y. Papakonstantinou. Hypothetical queries in an OLAP environment. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 220–231, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

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

[8] J. Carver, M. Fisher, II, and G. Rothermel. An empirical evaluation of a testing and debugging methodology for excel. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 278–287, New York, NY, USA, 2006. ACM.

[9] C. Chambers and M. Erwig. Reasoning about spreadsheets with labels and dimensions. *J. Vis. Lang. Comput.*, 21(5):249–262, Dec. 2010.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74, Mar. 1997.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

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

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

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

[15] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 136–145, New York, NY, USA, 2005. ACM.

[16] M. Fisher and G. Rothermel. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. *SIGSOFT Softw. Eng. Notes*, July 2005.

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

[18] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. Ajax: an extensible data cleaning tool. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, page 590, New York, NY, USA, 2000. ACM.

[19] L. Golab, H. Karloff, F. Korn, and D. Srivastava. Data auditor: exploring data quality and semantics using pattern tableaux. *Proc. VLDB Endow.*, 3(1-2):1641–1644, Sept. 2010.

[20] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In T. Ball and M. Sagiv, editors, *POPL*, pages 317–330. ACM, 2011.

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

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

[23] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 317–328. ACM, 2011.

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

[25] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 127–138, New York, NY, USA, 1995. ACM.

[26] T. Herndon, M. Ash, and R. Pollin. 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.

[27] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner. On the empirical evaluation of fault localization techniques for spreadsheets. In *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 68–82, Berlin, Heidelberg, 2013. Springer-Verlag.

[28] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

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

[30] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21:1–21:44, Apr. 2011.

[31] D. Luebbers, U. Grimmer, and M. Jarke. Systematic development of data mining-based data quality tools. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB '03, pages 548–559. VLDB Endowment, 2003.

[32] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

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

[34] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 302–312, New York, NY, USA, 2002. ACM.

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

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

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

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

[39] M. Sakal and L. Raković. Errors in building and using electronic tables: Financial consequences and minimisation techiques. *International Journal of Strategic Management and Decision Support Systems in Strategic Management*, 17(3):29–35, 2012.

[40] V. Samar and S. Patni. Controlling the information flow in spreadsheets. *CoRR*, abs/0803.2527, 2008.

[41] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, Apr. 2012.

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

Spreadsheet	Formulas	Cells	Cells	Runtime	Dep.	Impact	Impact
		raw	*weighted*	*total (s)*	**Analysis**	**Analysis**	**Scoring**
Inventory_Control	33	21	0	4.71	1.67	1.59	1.42
Logistikkostnader	73	29	26	8.88	3.48	2.97	2.40
HMWK112403	36	41	27	2.31	0.88	0.78	0.63
30day	125	92	30	3.01	1.39	1.31	0.27
Fin_info	25	40	32	2.75	0.082	2.65	0.01
2002fairreport	3	39	39	4.30	1.29	1.67	1.30
9620040303160820	42	81	81	4.77	1.19	2.74	0.81
Inventory errors	100	129	90	2.83	1.25	1.02	0.53
grades	227	661	96	154.45	3.06	149.85	1.51
expenses_ans	57	60	120	3.24	0.92	2.15	0.15
grades2002	61	143	123	2.67	1.03	1.11	0.51
Example_3	71	130	127	3.15	1.22	1.56	0.28
lmc_financial	72	148	142	17.15	4.69	7.63	4.80
104r	22	146	144	6.66	1.81	3.26	1.54
TRAIL INVENTORY N#A850A	2	156	156	6.15	1.12	3.99	0.99
Grades-6_excerpt	106	168	168	1.83	1.10	0.45	0.25
intresults	1066	3158	239	318.91	17.12	287.63	14.12
OakProducts	69	271	242	6.82	1.67	4.20	0.91
am_skandia_fin_supple#A80EE	56	272	268	6.64	1.53	4.01	1.06
E04_AppE_Census_Database_50	42	300	300	39.04	4.07	32.72	2.22
pfi-anxa	5	310	310	73.56	16.38	33.10	24.05
q exhibit54-OEA	797	1160	365	102.56	18.03	68.70	15.79
econ424-fall2003-publ#A8A23	93	517	384	62.83	3.91	56.96	1.93
Grades_EEE481&581	177	757	756	40.11	3.31	35.74	1.03
gpa_calculator	80	80	819	115.86	1.88	113.67	0.28
s446gradessp04	335	1369	1247	129.36	9.76	113.29	6.27
NEW	2626	2574	2403	683.32	115.75	440.30	127.23

Table 1. The benchmark suite of 30 spreadsheets, a random sample from the EUSES repository [16], ordered by weighted number of cells. The raw number of cells indicates the total number of cells that are used in any formula; the weighted number of cells weighs cells *in ranges* by the number of formulas that depend on it. A breakdown of CHECKCELL execution times (in seconds) appears on the right side.