Validating the Unit Correctness of Spreadsheet Programs[*]

Tudor Antoniu[†]
Blunk Microsystems

Paul A. Steckler[‡]
Northrop Grumman IT/FNMOC

Shriram Krishnamurthi
Brown University

Erich Neuwirth
Universität Wien

Matthias Felleisen
Northeastern University

Abstract

Financial companies, engineering firms and even scientists create increasingly larger spreadsheets and spreadsheet programs. The creators of large spreadsheets make errors and must track them down. One common class of errors concerns unit errors, because spreadsheets often employ formulas with physical or monetary units.

In this paper, we describe XeLda, our tool for unit checking Excel spreadsheets. The tool highlights cells if their formulas process values with incorrect units and if derived units clash with unit annotations. In addition, it draws arrows to the sources of the formulas for debugging. The tool is sensitive to many of the intricacies of Excel spreadsheets including tables, matrices, and even circular references. Using XeLda, we have detected errors in some published scientific spreadsheets.

1 Spreadsheet Programming

End users program. They program when they write database queries, when they design a style sheet for a word processor, or when they use a mail merge program to send a letter to a large group of people. Spreadsheet programming is the most common form of end user programming. People use spreadsheets for accounting, for financial modeling, for analysis and rapid prototyping in engineering, for teaching at all levels, and for many other purposes.

As people have become comfortable with spreadsheets, the complexity of spreadsheet programs has grown significantly. Whereas early spreadsheet programs consisted of a single worksheet for a simple accounting task, today financial experts, engineering firms and even scientists routinely use spreadsheets that consist of numerous worksheets encompassing hundreds and thousands of formulas. Indeed, these worksheets often depend on each other just like modules and procedures in large programs.

Like any other programs, spreadsheet programs suffer from errors. As Panko [17] points out, spreadsheet programmers often lack formal training and are therefore even more likely than ordinary programmers to make mistakes. By failing to fully appreciate the potential and uses of spreadsheet languages, the research community has largely abandoned these programmers; to this day, few researchers develop tools that help spreadsheet programmers check, test, and debug their spreadsheets. Notable exceptions include Burnett, Erwig and Rothermel [5, 6, 19].

In this paper we present XeLda, a new tool for validating the unit correctness of Excel spreadsheet programs. The widespread use of Excel in number-rich domains like engineering, finance and science, and the importance of unit checking in these applications makes this an important form of enhancing spreadsheet validity. Unit checking is also an important choice because widely available spreadsheets such as Excel do not support it.[1] Finally, we know from "standard" software engineering that unit errors such as those in the Mars lander can have costly consequences [1].

This paper's second section is a brief tutorial on the use of XeLda. In the third section, we explain how XeLda performs unit checking and how it copes with Excel's complexities, including tables, matrices, and circular dependencies among formulas. In the fourth section, we provide evidence that unit checking with XeLda is effective and necessary. The remaining sections cover related and future work.

2 A Brief XeLda Tutorial

Figure 1 shows the XeLda control panel, through which the user identifies a spreadsheet for unit-checking.[2] Pressing the `Load File` button starts an instance of Excel with

[*]Work partially supported by NSF grants ESI-0010064, ITR-0218973, SEL-0305949 and SEL-0305950.

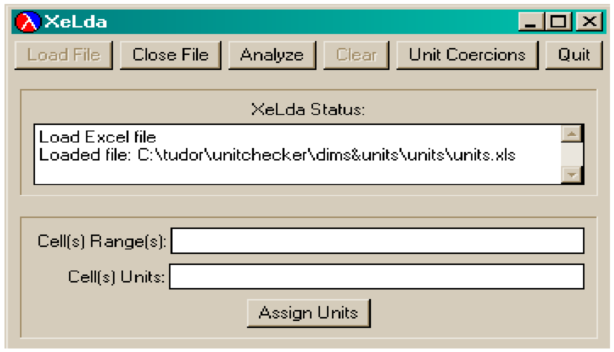
[†]Research performed at Brown University.

[‡]Research performed at Northeastern University.

[1]The `CONVERT` function of Excel converts numbers from one unit to another, but it does not perform unit *checking* over the spreadsheet.

[2]XeLda is a DrScheme [8] program. XeLda uses the MysterX extension to DrScheme [20] to communicate with Excel.

the specified spreadsheet. XeLda needs cells with numbers to be annotated with units; otherwise it treats them as dimensionless. XeLda then computes the unit for formula cells, and compares this against any annotation made by the user. Units are placed in Excel comment fields via Excel directly or using XeLda's control panel.



Load Excel file
Loaded file: C:\tudor\unitchecker\dims&units\units\units.xls

Figure 1. XeLda Control Panel

When the user requests an analysis (via the `Analyze` button), XeLda may find two kinds of errors:

- a derived unit doesn't match a cell's unit annotation (a *match error*), or
- a formula uses units in an inconsistent manner (a *consistency error*).

XeLda flags these errors by coloring the cells where they occur, orange for a match error and yellow for a consistency error. In addition, whenever an error occurs at a cell, all cells that depend on it are colored purple, indicating *error propagation*. (These choices are preliminary; user studies should reveal better mechanisms for highlighting errors.)

When a unit error occurs, it is useful to know *why* the error occurred, and what the *sources* of the error are. XeLda therefore gives an explanation of why an error has occurred in a cell by providing descriptive text in the cell's comment field. The sources of an error are shown by drawing arrows to the error cell from the cells that it depends on.[3] Figure 2 shows a XeLda-analyzed spreadsheet with a textual explanation display and all the source arrows drawn. Cell `B5`, annotated with `kg-m/s^2`, is the product of cells `A2` and `C2`, annotated with `kg` and `m-s^2`, respectively. Therefore, there is a mismatch between the unit computed for `B5` and its annotation. The other error occurs at cell `B12`, whose formula attempts to compare `apples` (from cell `A9`) with `oranges` (from cell `C9`). In both errors, the arrows indicate the sources of data for the erroneous cell. (Figure 5 presents a more detailed example of these arrows.)

[3]While we employ Excel's existing mechanism for presenting dependencies, we were inspired by similar ideas in the MrSpidey static debugger [9] and by spreadsheet visualization work [13].

3 Foundations

Spreadsheet cells contain formulas and unit annotations. The abstract syntax of *formulas* is:

$$e ::= n \mid \textit{cell-ref} \mid \textit{id} \mid e \ \textit{op} \ e \mid \textit{fun}(e, \ldots, e)$$

where n is a number, *cell-ref* is a cell reference, *op* is an arithmetic operator, and *fun* is an identifier denoting an Excel function. Examples of Excel functions are `SUM`, `AVERAGE`, and `MAX`. Cell references obey Excel's conventions (for example, `A1` and `B52`). An *id* is an identifier that names a spreadsheet cell. In the concrete syntax of formulas, parentheses may be used for grouping. This grammar purposely ignores a few constructs found in Excel formulas, such as boolean constants and conditionals, because they do not pose interesting problems.

Units represent the dimensions associated with a numeric value, consisting of a possibly empty list of unit-exponent pairs or an error:

$$\begin{aligned} U ::= & ((w\ n)\ldots) \mid \texttt{error/equality} \mid \\ & \texttt{error/propagate} \mid \texttt{error/circular} \end{aligned}$$

where each w is the name of a unit and n is an integer exponent. The names of units are arbitrary.[4] The empty list of units denotes dimensionlessness. A nonempty list of units and their exponents denotes a product of units; a negative exponent indicates division. For example, the unit

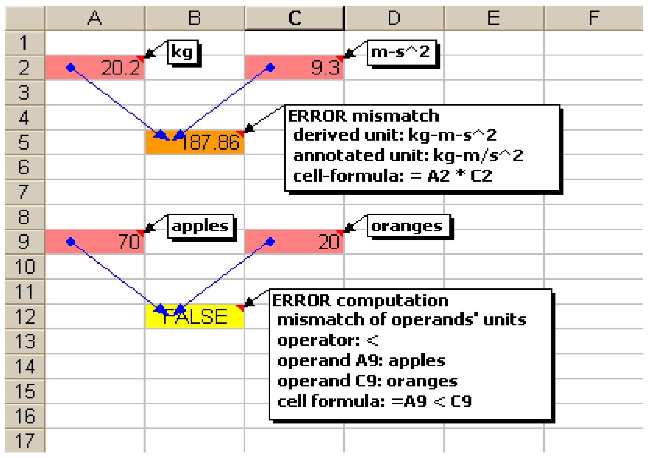
```
((kilogram 1) (meter 1) (second -2))
```

denotes an SI Newton. We use the notation $u_1 @ u_2$ to denote the multiplication (appending) of two units. The error `error/equality` corresponds to mismatches and inconsistency, and `error/propagate` ensues from propagation. We discuss `error/circular` in section 3.3.

Because we wish to compare units, it is convenient to have a normal form for them. The order of a compound unit's constituents is unimportant: a kilogram–meter denotes the same unit as a meter–kilogram. A unit name needs to appear within a unit only once, because the exponents in multiple occurrences can be summed. Therefore, we define a unit to be in *normal form* when:

- *each w in $w_1, \ldots, w_m$ is distinct;*
- *$w_i \leq w_{i+1}$ for $1 \leq i < m$, where the comparison on the w's is lexicographic; and*
- *$n_j \neq 0$ for $1 \leq j \leq m$.*

[4]XeLda does not restrict the set of unit names, so it can support domain-specific units or ones that come into existence over time (such as new currencies like the euro). XeLda does offer support for unit *coercion*; see section 3.4.



ERROR mismatch
derived unit: kg-m-s^2
annotated unit: kg-m/s^2
cell-formula: = A2 * C2

ERROR computation
mismatch of operands' units
operator: <
operand A9: apples
operand C9: oranges
cell formula: =A9 < C9

Figure 2. Unit Errors in a Spreadsheet

This definition is close to Kennedy's presentation of units as elements of an Abelian group [15], except for the sorting requirement, which is useful for an implementation. Clearly, we can obtain the normal form of any unit by summing exponents of like units, filtering out units with a zero exponent, and sorting on the unit names. We use $[u]$ to denote the normal form of u.

3.1 Calculating units

To compute units for each non-empty cell in a spreadsheet, we first partition the set of cells into those that contain a number (*value cells*) and those whose value is derived from a formula (*formula cells*).[5] For value cells, XeLda expects to find the unit in the cell's annotation; otherwise, it treats the number as a dimensionless constant. For formula cells, XeLda combines the units from the input cells according to the laws that govern the functions and operations of the underlying formula.[6] We elaborate on this strategy below, but intuitively, it corresponds to the way people derive units during manual calculations.

Except in the case of circular references, described separately in section 3.3, we use *unit transformers* to compute units. For each spreadsheet function we introduce a corresponding unit transformer, which consumes one or more units and produces a unit for the result. The unit transformer has the same arity as the Excel function it represents. To avoid ambiguity, we use a subscript to represent the Excel function ($+_{XL}$, $*_{XL}$, $\texttt{SUM}_{XL}$, $\texttt{AVERAGE}_{XL}$, etc.) and a hat superscript to denote its corresponding unit transformer (each Fun_{XL} yields a $\widehat{Fun}$).

Consider $+_{XL}$, which performs addition in Excel. We define the unit transformer of $+_{XL}$ as:

$$U_1 \,\widehat{+}\, U_2 = \begin{cases} \texttt{error/propagate} & \text{if } \textit{error-unit}(U_1) \text{ or } \textit{error-unit}(U_2) \\ U_1 & \text{if } U_1 = U_2 \\ \texttt{error/equality} & \text{otherwise} \end{cases}$$

where for any unit U, *error-unit*(U) holds iff U is an error unit. That is, the units associated with the arguments to $+$ must be identical and not error units; otherwise, we have a unit equality error. The $\widehat{-}$ unit transformer for subtraction is identical to $\widehat{+}$. The unit transformer for multiplication is:

$$U_1 \,\widehat{*}\, U_2 = \begin{cases} \texttt{error/propagate} \\ \quad \text{if } \textit{error-unit}(U_1) \text{ or } \textit{error-unit}(U_2) \\ [U_1 \; @ \; U_2] \\ \quad \text{otherwise} \end{cases}$$

The unit transformer for division has a slight twist:

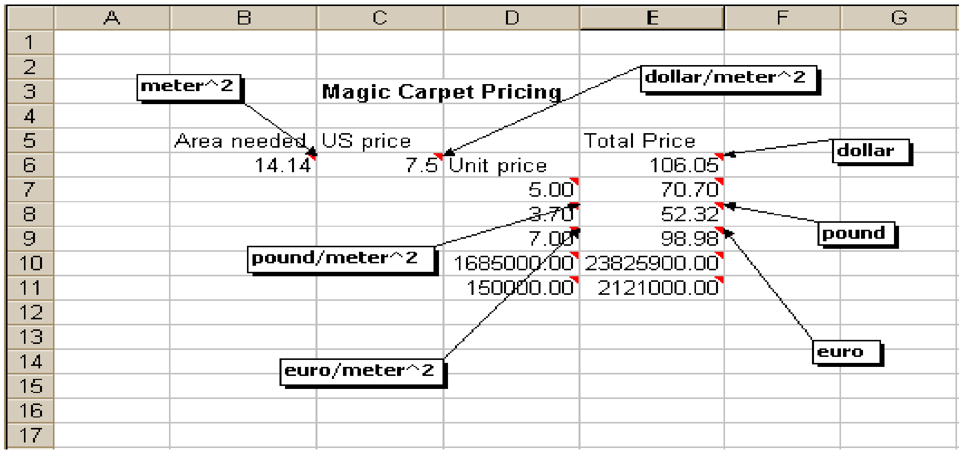
$$U_1 \,\widehat{/}\, U_2 = \begin{cases} \texttt{error/propagate} \\ \quad \text{if } \textit{error-unit}(U_1) \text{ or } \textit{error-unit}(U_2) \\ [U_1 \; @ \; \overline{U_2}] \\ \quad \text{otherwise} \end{cases}$$

where $\overline{U}$ is U with all the signs of exponents reversed. We similarly define transformers for other functions.

The unit for a formula is derived bottom-up, starting at value cells and propagating up to formula cells (treating unannotated value cells as dimensionless). For cells with formulas, we use the corresponding unit transformers to provide a derived unit. We handle cell references and identifiers naming cells in the same natural manner. After all units in a spreadsheet have been calculated, we compare them with the annotations on subformulas to detect mismatches.

[5] A formula consisting of just a number is treated as a value.

[6] The distinction in Excel between functions and operators is syntactic; from here on, we refer to both as *functions*.



	A	B	C	D	E	F	G
1							
2							
3			Magic Carpet Pricing				
4							
5		Area needed	US price		Total Price		
6		14.14	7.5	Unit price	106.05		
7				5.00	70.70		
8				3.70	52.32		
9				7.00	98.98		
10				1685000.00	23825900.00		
11				150000.00	2121000.00		
12							
13							
14							
15							
16							
17							

Figure 3. Data Table with Element-Polymorphic Table Map

3.2 Tables and matrices

Excel supports tables and matrix operations. Because Excel does not associate formulas directly with the result cells, XeLda must deal with these operations separately.

3.2.1 Tables

A *data table* in Excel is a range of cells that shows the results for one formula based on different input values. That is, tables are produced from ranges of cells much as functional programmers derive lists by mapping functions over input lists. Using tables, one can calculate multiple variations of an operation and view them as a block of cells. Tables are thus especially useful for "what-if" calculations.

Figure 3 shows a simple table that computes the cost of a magic carpet in several currencies. The original input to the table is the price of a magic carpet in dollars per square meter (`C6`). The "what if" inputs to the table are prices per square meter, where the price varies by country (`D7:D11`). In the resulting table (`E7:E11`), each numeric price is annotated with a unit indicating the appropriate currency.

When performing table operations, Excel uses the layout of the spreadsheet to determine what input values to use and where to put the results. Concretely, cell `E6` contains the formula `B6 * C6`, so its value is 106.05. The range `D7:D11` contains some numbers. Each cell in the range `E7:E11` contains the formula {`=TABLE(,C6)`}. The comma before the cell name indicates that the replacement values for `C6` come from the column to the left. Excel "maps" the formula of `E6`, replacing `C6` in the formula, over the values in `D7:D11` and places each result in the adjacent cell in column `E`.

For each cell in a table, we compute a unit by applying the unit transformer for the cell with the formula, except that we use the unit for the cell pointwise instead of that for the original input. For each input element, we always use the same unit transformer; hence the computation of units for tables yields polymorphism over units. To continue the analogy with mapping over lists, our approach to tables is comparable to homologously mapping a function over a heterogeneously-typed list and obtaining heterogeneously-typed results.

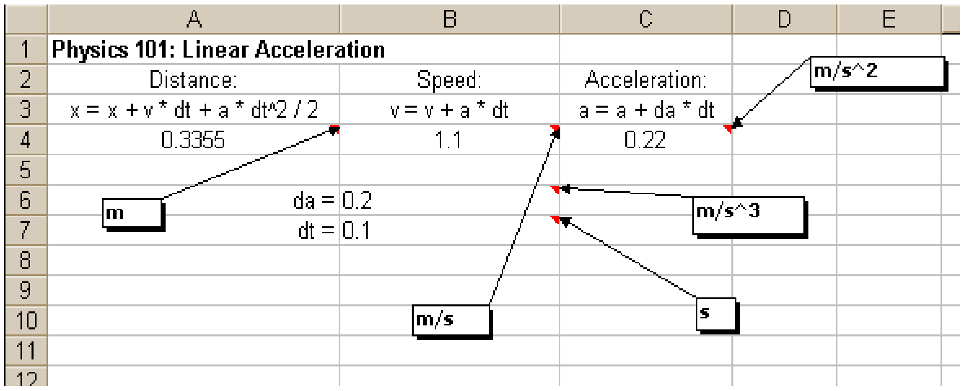
3.2.2 Matrix values

In Excel, a matrix value occupies a rectangular block of cells. Each cell in the matrix contains the formula that produced the result. In the case of matrix multiplication, that formula has the form {`=MMULT(M1,M2)`} where `M1` and `M2` are cell ranges denoting matrices.

For each cell in the result matrix, we check that all elements in the row in `M1` required for the calculation of that cell have equal units. Similarly, we check the units for the column in `M2` that produced the entry for equality. If any of those input elements has an error unit, the unit for the result cell is `error/propagate`. If either of the equality checks fails, the result is `error/equality`. Otherwise, the unit for the cell becomes the normalized product of the units for the row in `M1` and the column in `M2`.

3.3 Circular references

Excel formulas can depend on themselves (through direct or indirect references). When a user enters such formulas, Excel issues a warning. If the user wishes to proceed anyway, Excel computes a solution iteratively; unless otherwise specified, cells are initialized with 0. A user-selectable limit on iteration enforces termination, even in the absence of a fixpoint. Circular references are useful for representing many physical and economic models that involve feedback



	A	B	C	D	E
1	**Physics 101: Linear Acceleration**				
2	Distance:	Speed:	Acceleration:		
3	x = x + v * dt + a * dt^2 / 2	v = v + a * dt	a = a + da * dt		
4	0.3355	1.1	0.22		
5					
6	da =	0.2			
7	dt =	0.1			
8					
9					
10					
11					
12					

Figure 4. Equations of Motion

loops, such as systems of differential equations and recurrence relations.

Figure 4 shows a spreadsheet containing three circular references. Its purpose is to iteratively compute the position of a particle under a linearly-increasing acceleration, given an initial position, velocity, and acceleration. Cells B6 and B7 specify the increments for acceleration and time, respectively. With each iteration, we compute a new position (cell A4), velocity (B4), and acceleration (C4). By varying the iteration limit in Excel, we can vary the time interval used to compute the new values.

XeLda is able to validate unit annotations in spreadsheets with circular formulas. While parsing, XeLda distinguishes formulas with circular dependencies from those with purely tree-structured dependencies. We first resolve units for the latter class of formulas, using the approach in section 3.1. We then use the results to derive units for the formulas with circular dependencies.

For formulas with circular references, unit derivation is a three-step process. First, for each circular formula, we generate a set of *constraints* containing *unit variables* and units. Second, we build equivalence classes of unit variables and propagate class representatives to other constraints. We are left with constraints that we can now map into algebraic equations, which are then transformed into a set of homogeneous linear equations. Finally, we solve for the unit variables using Gaussian elimination, yielding units for the circular formulas.

3.3.1 Constraint generation

When formulas have circular dependencies, we cannot apply unit transformers because dependency loops would lead to divergence. We instead generate constraints on units appropriate to that function. Essentially, constraint generation postpones application of unit transformers.

The constraint generation process produces units that contain only the units appearing in their inputs (excepting error units). In particular, both $\widehat{+}$ and $\widehat{-}$ impose *equality constraints* on their inputs, while $\widehat{*}$ and $\widehat{/}$ specify how to combine input units to obtain a result unit yielding *append constraints*. Other unit transformers may specify both equality and append constraints.

Within each formula containing a circular reference, for each use of an Excel function, we provide fresh unit variables for the application node and its arguments and generate constraints. For the arithmetic operators, we generate constraints as follows:

$$e_1 +_{XL} e_2 \quad \Rightarrow \quad \begin{cases} \alpha = \alpha_{e_1} \\ \alpha = \alpha_{e_2} \end{cases}$$

$$e_1 -_{XL} e_2 \quad \Rightarrow \quad \begin{cases} \alpha = \alpha_{e_1} \\ \alpha = \alpha_{e_2} \end{cases}$$

$$e_1 *_{XL} e_2 \quad \Rightarrow \quad \{ \; \alpha = [\alpha_{e_1} \mathbin{@} \alpha_{e_2}]$$

$$e_1 /_{XL} e_2 \quad \Rightarrow \quad \{ \; \alpha = [\alpha_{e_1} \mathbin{@} \overline{\alpha_{e_2}}]$$

where α is the unit variable associated with the application node and the α_{e_i} are the newly-generated unit variables for the formula's arguments. We also generate equality constraints for references to annotated value cells and to cells with noncircular formulas, whose units have already been computed. Hence, equality constraints can have nonvariable units on their right-hand sides.

Consider the spreadsheet in Figure 4. Cell C4 contains the circular formula: C4+B6∗B7. The constraints generated from this formula are:

$$\begin{aligned} \alpha_{\texttt{C4}} &= \alpha_1 \\ \alpha_{\texttt{C4}} &= \alpha_2 \end{aligned}$$

where α_1 and α_2 are fresh variables for the operands of $+$.

For the subformulas, we obtain:

$$\begin{array}{rcl}\alpha_1 & = & \alpha_{\texttt{C4}} \\ \alpha_2 & = & [\alpha_3 \mathbin{@} \alpha_4] \\ \alpha_3 & = & ((\texttt{m}\ 1)\ (\texttt{s}\ -3)) \\ \alpha_4 & = & ((\texttt{s}\ 1))\end{array}$$

where α_3 and α_4 are associated with the $*$'s arguments.

3.3.2 Constraint simplification

In order to solve the constraints, we use the following algorithm. From the equality constraints, generate equivalence classes of variables. For each class, if at least two members participate in constraints with distinct right-hand side units, choose `error/equality` as the *representative unit* for the class. If there is exactly one unit associated with the class, choose that unit as the representative unit. Otherwise, there is no representative unit for the class, so choose a representative variable for that class. Substitute the representative unit, if any, for occurrences of class members on the right-hand sides of append constraints; otherwise substitute the representative variable. Substitute the representative unit or variable for occurrences of class members on the left-hand sides of the append constraints.

Next, we deal with the append constraints. Each of these constraints has the form:

$$\alpha_i \quad = \quad [\beta_1 @ \beta_2] \tag{1}$$

where β is a metavariable ranging over unit variables and units.

A unit U may be written in algebraic form. Without loss of generality, suppose U is in normal form. Then its algebraic form is given by:

$$u \quad = \quad \prod_i w_i^{n_i} \tag{2}$$

where each w_i is a distinct unit name; in this algebraic setting, consider these to be constants.

From equation 1, we can have unit variables, units, or both, on the right-hand sides of append constraints in the place of the β_i. We can therefore represent every append constraint in the following algebraic forms:

$$\alpha \quad = \quad \alpha_1 \times \alpha_2 \tag{3}$$
$$\alpha \quad = \quad \alpha_1 \times u_1 \tag{4}$$
$$\alpha \quad = \quad u_1 \times u_2 \tag{5}$$

Because normalization is a syntactic, rather than semantic, issue, it does not appear in the algebraic representation. The $\times$ operator is commutative, so the order of its arguments does not matter.

3.3.3 Constraint solving

We wish to solve for the unit variables in terms of units. In one special case, we simplify before proceeding to Gaussian elimination. If the left-hand side unit variable also occurs on the right-hand side of equations of type 4, we can immediately deal with the constraint by examining the right-hand side unit u. If u is the empty list, we discard the constraint, because there is no effective constraint on the variable. Otherwise, associate `error/circular` with the variable, because there is no solution for the constraint.

For the remaining constraints, we divide through their left-hand sides. The equations 3–5 become:

$$1 \quad = \quad \alpha^{-1} \times \alpha_1 \times \alpha_2 \tag{6}$$
$$1 \quad = \quad \alpha^{-1} \times \alpha_1 \times u_1 \tag{7}$$
$$1 \quad = \quad \alpha^{-1} \times u_1 \times u_2 \tag{8}$$

Taking logarithms, we get:

$$0 \quad = \quad -\log\alpha + \log\alpha_1 + \log\alpha_2 \tag{9}$$
$$0 \quad = \quad -\log\alpha + \log\alpha_1 + \log u_1 \tag{10}$$
$$0 \quad = \quad -\log\alpha + \log u_1 + \log u_2 \tag{11}$$

From equation 2, by taking the logarithm of a unit in algebraic form u, we have:

$$\log u = \sum_i (n_i \times \log w_i)$$

Substituting for the logarithms of units in equations 9 – 11, we obtain the following *linear equations*:

$$0 \quad = \quad -\log\alpha + \log\alpha_1 + \log\alpha_2 \tag{12}$$
$$0 \quad = \quad -\log\alpha + \log\alpha_1 + \sum_i (n_{1_i} \times \log w_{1_i}) \tag{13}$$
$$0 \quad = \quad -\log\alpha + \sum_i (n_{1_i} \times \log w_{1_i}) + \sum_j (n_{2_j} \times \log w_{2_j}) \tag{14}$$

We solve these equations for the $\log\alpha$'s using Gaussian elimination. If we have fewer equations than variables, we attempt to solve for as many variables as possible. The remaining variables are unconstrained, so we assign them `error/circular`.

For each unit variable α that may have a solution, Gaussian elimination produces equations of the form:

$$\log\alpha = \sum_i \left(\log w_i \times \frac{n_i}{c}\right)$$

where c is a nonzero integer. Equivalently:

$$\alpha = \prod_i w_i^{n_i/c}$$

We accept only solutions where all n_i/c are integers. In all other cases, we assign α the unit `error/circular`.

Let us illustrate how this approach applies to our Figure 4. After constraint generation and equivalence class substitution we have the following append constraints to solve:

$$\begin{aligned}\alpha_{\mathtt{A4}} &= [\alpha_{\mathtt{B4}} \mathbin{@} ((\mathtt{s}\ 1))] \\ \alpha_{\mathtt{A4}} &= [((\mathtt{m}\ 1)\ (\mathtt{s}\ {-2})) \mathbin{@} ((\mathtt{s}\ 2))] \\ \alpha_{\mathtt{B4}} &= [((\mathtt{m}\ 1)\ (\mathtt{s}\ {-2})) \mathbin{@} ((\mathtt{s}\ 1))]\end{aligned}$$

Converting these to algebraic equations we obtain:

$$\begin{aligned}\alpha_{\mathtt{A4}} &= \alpha_{\mathtt{B4}} \times \mathtt{s} \\ \alpha_{\mathtt{A4}} &= \mathtt{m} \\ \alpha_{\mathtt{B4}} &= \mathtt{m} \times \mathtt{s}^{-1}\end{aligned}$$

We have one equation of type 4 and two of type 5. Following the steps outlined above, we get the system of linear equations

$$\begin{aligned}0 &= -\log \alpha_{\mathtt{A4}} + \log \alpha_{\mathtt{B4}} + \log \mathtt{s} \\ 0 &= -\log \alpha_{\mathtt{A4}} + \log \mathtt{m} \\ 0 &= -\log \alpha_{\mathtt{B4}} + \log \mathtt{m} - \log \mathtt{s}\end{aligned}$$

By Gaussian elimination we have:

$$\begin{aligned}\log \alpha_{\mathtt{A4}} &= \log \mathtt{m} \\ \log \alpha_{\mathtt{B4}} &= \log \mathtt{m} - \log \mathtt{s}\end{aligned}$$

hence

$$\begin{aligned}\alpha_{\mathtt{A4}} &= \mathtt{m} \\ \alpha_{\mathtt{B4}} &= \mathtt{m}/\mathtt{s}\end{aligned}$$

as desired.

We choose to not use this general technique for non-circular formulas partly because it is more computationally expensive, but mainly because it would produce poorer error reports. Using constraint solving in the presence of equational reasoning makes it difficult to provide a "direction" for the error (as observed by many in the ML community [21]), whereas the technique in section 3.1 does not suffer from this problem.

3.4 Unit coercions

Because Excel does not understand units, it can provide unexpected answers. Suppose cells `A1` and `A2` both contain 5 and cell `A3` contains the formula `=A1 + A2`. Then `A3` always shows the value 10, even if `A1`'s annotation is `feet` and `A2`'s is `meters`.

In contrast, XeLda as presented would flag an error because the units of the summands don't match. We address this shortcoming via unit coercions. The `Unit Coercions` button of the control panel presents a window in which users can specify coercions. XeLda adapts the union-find algorithm to create equivalence classes of interconvertible units, reporting errors on encountering inconsistent coercions. XeLda then rewrites Excel formulas to reflect these coercions. Thus, if the user had specified that one `meters` were equal to 3.3 `feet`, XeLda would rewrite the formula in `A3` as `=A1 + (3.3 * A2)`.

After performing the unit conversions, XeLda presents the answer in terms of the canonical unit chosen for each equivalence class of units. To change the presentation of the result, the user annotates the formula cell with a unit; XeLda not only verifies this annotation, but also treats it as a presentation directive. In the example above, for instance, XeLda has chosen `feet` as the representative unit, but annotating cell `A3` with `meters` forces XeLda to rewrite the formula as `=1/3.3 * (A1 + (3.3 * A2))`, so the user sees the computed answer as 6.515152 (`meters`).

3.5 Combining features

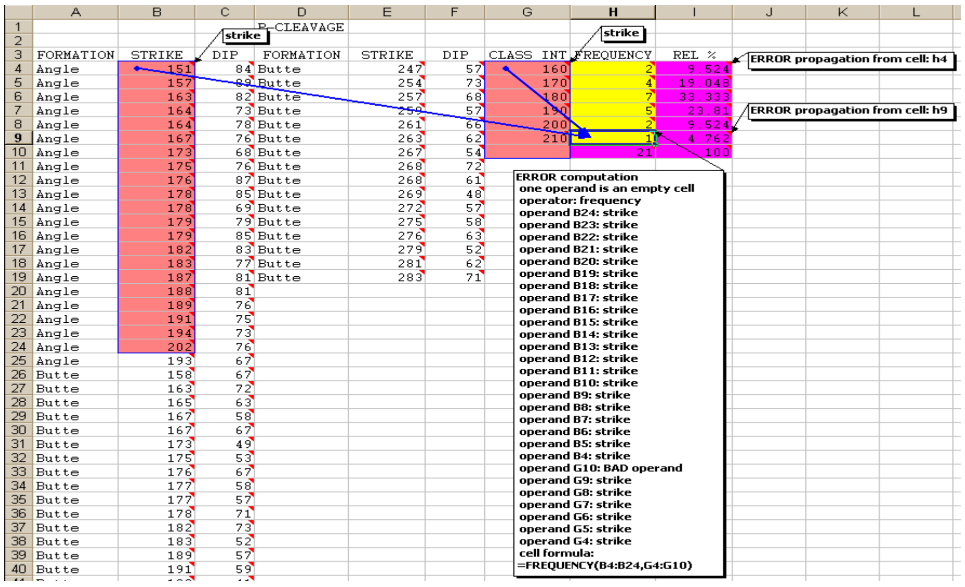
XeLda's unit checking operations are designed and implemented in an orthogonal manner. Thus, if a spreadsheet uses tables with formulas that employ circular references, XeLda can still validate its unit annotations. Similarly, if such annotations are wrong, it can still identify the erroneous cells.

4 Applying XeLda: A Case Study

We conducted a study to assess XeLda's ability to find errors. To this end, we applied XeLda to the scientific computing spreadsheets that accompany Filby's book [7]. Many of these spreadsheets specify units in textual headers for columns and rows containing numeric data. Using XeLda's unit annotation feature, we inserted units in the cells labelled by those headers.

When applying XeLda to these and to other existing spreadsheets, we found the unit annotation process involved relatively little time and effort. Of course, this does not immediately indicate how much effort a typical end-user would have to expend. On the one hand they know less about XeLda; on the other hand, their expertise in the domain will often be much greater than ours. Furthermore, Rothermel, et al. [23] provide empirical results that establish the value of a curiosity-centered approach in getting end users to enter assertions in spreadsheets created in the Forms/3 spreadsheet language.

The table in Figure 6 describes the spreadsheets from Filby's book that we used to test XeLda. Each horizontal grouping represents one Excel file; within each grouping, each line represents a worksheet. The size given is the number of non-empty cells. The first time column presents overall analysis (wall clock) time; the next two divide this between the time spent in communication with Excel (through



	A	B	C	D	E	F	G	H	I
1				P-CLEAVAGE					
2									
3	FORMATION	STRIKE	DIP	FORMATION	STRIKE	DIP	CLASS INT	FREQUENCY	REL %
4	Angle	151	84	Butte	247	57	160	2	9.524
5	Angle	157	89	Butte	254	73	170	4	19.048
6	Angle	163	82	Butte	257	68	180	7	33.333
7	Angle	164	73	Butte	259	57	190	5	23.81
8	Angle	164	78	Butte	261	66	200	2	9.524
9	Angle	167	76	Butte	263	62	210	1	4.762
10	Angle	173	68	Butte	267	54		21	100
11	Angle	175	76	Butte	268	72			
12	Angle	176	87	Butte	268	61			
13	Angle	178	85	Butte	269	48			
14	Angle	178	69	Butte	272	57			
15	Angle	179	79	Butte	275	58			
16	Angle	179	85	Butte	276	63			
17	Angle	182	83	Butte	279	52			
18	Angle	183	77	Butte	281	62			
19	Angle	187	81	Butte	283	71			
20	Angle	188	81						
21	Angle	189	76						
22	Angle	191	75						
23	Angle	194	73						
24	Angle	202	76						
25	Angle	193	67						
26	Butte	158	67						
27	Butte	163	72						
28	Butte	165	63						
29	Butte	167	58						
30	Butte	167	67						
31	Butte	173	49						
32	Butte	175	53						
33	Butte	176	67						
34	Butte	177	58						
35	Butte	177	57						
36	Butte	178	71						
37	Butte	182	73						
38	Butte	183	52						
39	Butte	189	57						
40	Butte	191	59						

strike

strike

ERROR propagation from cell: h4

ERROR propagation from cell: h9

ERROR computation
one operand is an empty cell
operator: frequency
operand B24: strike
operand B23: strike
operand B22: strike
operand B21: strike
operand B20: strike
operand B19: strike
operand B18: strike
operand B17: strike
operand B16: strike
operand B15: strike
operand B14: strike
operand B13: strike
operand B12: strike
operand B11: strike
operand B10: strike
operand B9: strike
operand B8: strike
operand B7: strike
operand B6: strike
operand B5: strike
operand B4: strike
operand G10: BAD operand
operand G9: strike
operand G8: strike
operand G7: strike
operand G6: strike
operand G5: strike
operand G4: strike
cell formula:
=FREQUENCY(B4:B24,G4:G10)

Figure 5. Unit Error in Off-the-Shelf Spreadsheet

COM's direct interfaces) and that consumed by the actual analysis, respectively.[7] Importantly, the analysis time is extremely small in most cases, which means an interface wrapper with minimal overhead would make XeLda usable for prototyping and incremental development. The last column indicates whether XeLda found any errors.

Given annotated spreadsheets, we are interested primarily in the rate of false negatives, because flagging correct spreadsheets as incorrect would reduce the tool's effectiveness from a user's standpoint. Since these spreadsheets came from a published text, we did not expect to find errors; we originally ran these tests purely to measure performance on useful spreadsheets. We were surprised when XeLda reported unit errors in three of the worksheets, but hand-examination revealed that all three were actual errors, not false negatives intoduced by XeLda. (We also conducted less rigorous testing by seeding spreadsheets with errors, and XeLda successfully detected all of them with neither false positives nor false negatives. However, we need a more thorough evaluation than the one described here before we can make stronger claims about effectiveness.)

For the "Oscillation" worksheets, the errors were caused by inappropriate textual labelling of units by the author. The "Cleavage" error was caused by supplying too big a cell range to the Excel `FREQUENCY` function. Figure 5 shows that spreadsheet. The `FREQUENCY` function is used in the formula for each of the cells in the range `H4:H9`. That function takes two vectors of numbers, where the second is in increasing order, indicating bins in which to place numbers from the first vector. It returns a vector that contains the number of numbers from the first vector within each bin; the last element in the returned vector is the number of numbers greater than the highest bin. In the spreadsheet shown, the cell range for the second argument erroneously includes the cell `G10`, which has been left blank and has no unit annotation. All other values in column `G` have the unit `strike`; because that does not agree with the unit for `G10`, there is an error in the shaded cells in column `H`.

5 Related Work

The most closely related work on detecting errors in spreadsheets is by Erwig and Burnett [5, 6]. Their "units" are taken from the names of row and column headers in spreadsheets. While this offers the promise of avoiding the burden of user annotation, they do not describe an effective technique for performing this unit inference (which appears to require intelligence to determine the relationship between units). In addition, unit inference may depend heavily on the structure of the spreadsheet, and would therefore be extremely sensitive to small changes in it. Nevertheless, a successful implemenation of unit annotation could seed XeLda also. Erwig and Burnett briefly describe an implementation [5], but do not provide any validation. Ahmad, et al. [2] provide a critique of the sensitivity of Erwig and Burnett's

[7]The time is lost to inter-process communication between XeLda and Excel. The use of .NET's interfaces should shrink this time. Indeed, Joe Marshall of Northeastern has produced a prototype implementation atop .NET. Unfortunately, .NET currently uses COM internally [confirmed in personal communication, 2003-09-13, by Erik Meijer of Microsoft], so we cannot yet evaluate the prototype for performance improvements. Other strategies would be to re-implement XeLda entirely in Visual Basic, which would add some cost to the analysis time, or to port XeLda to Excel's XLL API for plugins.

Author	Description	Size (cells)	Time (mm:ss)	COM (mm:ss)	Analysis (mm:ss)	Error?
S. Leharne	Acid Base Titration	109	0:23	0:22	0:01	
W.J. Orvis	Oscillations Frequency	43	0:21	0:17	0:04	✓
	Oscillations Euler Method	345	2:21	1:35	0:46	✓
A.A. Gorni	Cubic Crystalline Systems X-Ray Diffraction	83	0:46	0:43	0:03	
W.J. Orvis	Electron Drift Velocity in GaAs	44	0:15	0:13	0:02	
J.P. LeRoux	Cleavage Strike Direction	236	1:09	1:06	0:03	✓
	Palaeocurrent	284	1:12	1:08	0:04	
	Untilt	53	0:17	0:15	0:02	
	Chi-square	41	0:05	0:04	0:01	
A.A. Gorni	Grain size of microstructure	40	0:13	0:12	0:01	
E. Neuwirth	Feigenbaum Diagram	1000	2:09	2:08	0:01	
E. Neuwirth	Simple Model	54	0:04	0:03	0:01	
	Parametric Model	55	0:07	0:06	0:01	
	Complex Model	56	0:06	0:05	0:01	
	Complex Model with Table	75	0:10	0:09	0:01	
	Complex Model with Stepwidth	57	0:09	0:08	0:01	
	Volterra-Lotka Model	1983	16:28	14:03	2:25	
	Planets	4001	8:37	8:24	0:13	
	Planets Halfstep	4001	8:30	8:18	0:12	
W.J. Orvis	Blackbody spectral emission	507	0:35	0:34	0:01	
A.A. Gorni	Viscometric molecular weight	41	0:45	0:44	0:01	
A.A. Gorni	Point count method	26	0:16	0:15	0:01	

Figure 6. Experimental Results

type system, use this to improve the type system by including both “is-a” and “has-a” relationships, implement the refined version and discuss validation, but their approach is still fundamentally tied to a notion of types linked to spreadsheet structure.

Other work on the detection of errors in spreadsheets uses approaches radically different from ours (and thus catches fairly different forms of errors). Burnett and colleagues have also explored the idea of placing user annotations on cells, though not unit annotations, to improve the correctness of spreadsheets [4]. Rothermel, et al. [19] identify spreadsheet users’ manually seeded errors in cell expressions through the use of data flow adequacy criteria and coverage monitoring. Their solution, which is incremental, supplements the immediate visual feedback that a spreadsheet language presents by providing additional feedback about “testedness”. Their empirical data demonstrate the high effectiveness of their technique: 81% of the faults in their test suite were detected. Their spreadsheets are, however, created in the Forms/3 spreadsheet language [3]. While languages such as Forms/3 are elegant and point the way to future spreadsheet designs, tools built around them are likely to have less immediate impact; they may also be somewhat less challenging to construct than tools that need to address the complexities of existing and popular tools such as Excel.

Many researchers have suggested adding units to programming languages, with Gehani [10] and House [12] among the pioneers. Kennedy described integrating units into ML [14] and a System F-like language [15]. Like these systems, XeLda implements unit polymorphism. XeLda only works with the fixed set of functions provided by Excel, not user-defined functions. However, whereas unification in ML-like languages restricts mapping operators to work on homogeneous aggregates, XeLda’s mapping operator is polymorphic over the possibly-heterogeneous units of elements in aggregates. Other researchers have also proposed unit extensions to ML and the lambda calculus, including Goubault [11], who proposed the use of a sorted type algebra that allows rational dimension exponents; Wand and O’Keefe [22], who allow programmers to add new units within a delimited scope; and Rittri, who considered dimensional analysis in the presence of polymorphic recursion [18]. Novak [16] has worked on a unit extension to GLISP that checks unit conversions and simplifies them through dimensional analysis.

6 Beyond a Prototype

We have designed and implemented a unit-checker for Microsoft Excel that is able to handle its complex idioms. XeLda was able to find unit errors in off-the-shelf spreadsheets, validating our effort beyond our expectations.

At this point, XeLda is a prototype. To apply XeLda to a broader family of practical spreadsheets, we will need several kinds of improvements. First, we need to design unit transformers and constraint generators for all Excel functions. There are over 300 such functions in Excel 2002, although many do not operate on numbers. One class of interesting Excel functions operates on relational databases embedded in spreadsheets. For example, the `DAVERAGE` function takes a range of cells representing the data, a column name, and another range that specifies query-by-example criteria; it returns the numeric average of the cells meeting the criteria. Hence, all cells in the named column should have the same units. Second, we need to conduct user studies to determine the utility of such an approach, and to design an effective interface. Third, we must address Excel macros. Fourth, we should add features based on properties of known units, such as those of the English and SI systems, to assist users in specific domains (such as science). Finally, XeLda's use of COM makes its access of spreadsheet data slow. When .NET's interfaces eliminate the inter-process communication overhead, XeLda will become considerably faster. In the meanwhile, we could obtain greater speed by using Excel's XLL interface for plugins.

Acknowledgment

Dr. Arnold Proulx of Draper Laboratories helped us cast append constraints as a system of linear equations. Michael Ernst provided valuable editing suggestions on short order.

References

[1] JPL Mars program. Mars Climate Orbiter Failure Board releases report, numerous NASA actions underway in response. Press release 99-134, Nov. 1999.

[2] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A type system for statically detecting spreadsheet errors. In *18th IEEE Int. Conf. on Automated Software Engineering*, 2003.

[3] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J.Rechwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 10:155–206, 2001.

[4] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *25th Intl. Conf. on Software Engineering*, 2003.

[5] M. Burnett and M. Erwig. Visually customizing inference rules about apples and oranges. In *2nd IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pages 140–148, 2002.

[6] M. Erwig and M. Burnett. Adding apples and oranges. In *Intl. Symp. on Practical Aspects of Declarative Languages*, volume 2257, pages 173–191. Springer, 2002.

[7] G. Filby, editor. *Spreadsheets in Science and Engineering*. Springer, 1995.

[8] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A progamming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[9] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proc. ACM SIGPLAN Conf. on Programming language design and implementation (PLDI '96)*, pages 23–32, 1996.

[10] N. Gehani. Units of measure as a data attribute. *Computer Languages*, 2:93–111, 1977.

[11] J. Goubault. Inférence d'unités physiques en ML. *Journées Francophones des Langages Applicatifs*, pages 3–20, 1994.

[12] R. House. A proposal for an extended form of type checking of expressions. *Computer Journal*, 26(4):366–374, 1983.

[13] T. Igarashi, J. D. Mackinlay, B. W. Chang, and P. T. Zellweger. Fluid visualization for spreadsheet structures. In *IEEE Symp. on Visual Languages*, pages 118–125, Sept. 1998.

[14] A. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, April 1996.

[15] A. Kennedy. Relational parametricity and units of measure. In *ACM Symp. on Principles of Programming Languages*, pages 442–455, 1997.

[16] G. S. Novak. Conversion of units of measure. *IEEE Trans. Software Engineering*, 21(8):651–661, 1995.

[17] R. R. Panko. What we know about spreadsheet errors. *J. End User Computing*, 10(2):15–21, Spring 1998.

[18] M. Rittri. Dimension inference under polymorphic recursion. In *Functional Programming and Computer Architecture*, pages 147–159, 1995.

[19] G. Rothermel, M. Burnett, L. Li, C. Dupui, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10:110–147, 2001.

[20] P. A. Steckler. Component support in PLT Scheme. *Software–Practice and Experience*, 32:933–954, 2002.

[21] M. Wand. Finding the source of type errors. In *ACM Symp. on Principles of Programming Languages*, pages 38–43, 1986.

[22] M. Wand and P. O'Keefe. Automatic dimensional inference. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: in honor of J. Alan Robinson*, pages 479–486. MIT Press, 1991.

[23] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. Harnessing curiosity to increase correctness in end-user programming. *ACM Conference on Human Factors in Computing Systems*, pages 305–312, 2003.