Intelligently Creating and Recommending Reusable Reformatting Rules

Chris Scaffidi, Brad Myers, Mary Shaw
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA
{cscaffid, bam, mary.shaw}@cs.cmu.edu

ABSTRACT

When users combine data from multiple sources into a spreadsheet or dataset, the result is often a mishmash of different formats, since phone numbers, dates, course numbers and other string-like kinds of data can each be written in many different formats. Although spreadsheets provide features for reformatting numbers and a few specific kinds of string data, they do not provide any support for the wide range of other kinds of string data encountered by users. We describe a user interface where a user can describe the formats of each kind of data. We provide an algorithm that uses these formats to automatically generate reformatting rules that transform strings from one format to another. In effect, our system enables users to create a small expert system called a "tope" that can recognize and reformat instances of one kind of data. Later, as the user is working with a spreadsheet, our system recommends appropriate topes for validating and reformatting the data. With a recall of over 80% for a query time of under 1 second, this algorithm is accurate enough and fast enough to make useful recommendations in an interactive setting. A laboratory experiment shows that compared to manual typing, users can reformat sample spreadsheet data more than twice as fast by creating and using topes.

Author Keywords

End-user programming, spreadsheets, consistent data format

ACM Classification Keywords

D.2.6 [Programming Environments]: Interactive environments

INTRODUCTION

Information workers commonly use spreadsheets to gather, analyze, and organize data [5]. The data is not always numerical or computational. In fact, one study found that over 40% of spreadsheet cells contain textual data rather than numbers or formulas [4]. The data includes email addresses, person names, city names, and URLs. Many other kinds of data are organization-specific, such as building names (and their abbreviations), project numbers, and accounting codes.

However, a dataset's usefulness is often limited by typos and formatting inconsistencies. For example, most cells in a spreadsheet column might contain correct American phone numbers, but some cells might omit the area code. Moreover, while most cells might be formatted like "888-555-1212", a few might be written in internationalized format. Typos and formatting irregularities can easily accumulate when workers copy and paste data from multiple web sites or other sources into a spreadsheet.

Because of typos and formatting inconsistencies, information workers often must review and correct datasets prior to sharing them or generating reports [5]. This review is tedious, error-prone, and mostly manual, as spreadsheets offer no automatic mechanisms for validating strings. Even after the user manually identifies cells that need fixing, actually correcting them requires manual editing, as spreadsheets can only reformat a small fixed set of types (such as dates).

Since many kinds of data are uncommon or organization-specific, tool designers cannot anticipate every kind of data that users will want to reformat. Thus, there is a need for a user-extensible model for string data. Our "topes" model fills this role [14]. Each "tope" in this model is a user-defined abstraction that contains functions for recognizing and reformatting a particular kind of data, such as URLs or project numbers. In essence, each tope is a small, user-defined expert system tailored to one kind of data.

Previously, we have shown that topes can validate many kinds of data in spreadsheets and web forms [14]. However, topes created in previous versions of our tope editing tool were somewhat brittle, in that modifying formats often broke the reformatting rules that joined one format to another. In this paper, we describe a new tool called Toped^{++} that automatically implements reformatting rules and repairs them as needed as users edit formats. By performing this work automatically, Toped^{++} greatly reduces the time and effort required for a user to "teach" the computer how to recognize and reformat each kind of data. A laboratory experiment shows that compared to manual typing, users can reformat sample spreadsheet data more than twice as fast by creating and using topes, with an uncorrected error rate of approximately 1 in 1000. On average, once a user has created a tope and used it to validate and reformat 47 strings, the effort of creating the tope would have "paid off". Moreover, the tope could be reused later to validate and reformat an unlimited number of future spreadsheets.

IUI 2009, Feb 8-11, 2009, Sanibel, FL, USA.

To reuse a tope, users must be able to find a relevant tope for the data at hand. We describe a heuristic-based "search-by-match" algorithm to find topes that match example strings. For example, this algorithm could quickly locate topes that match "888-555-1212". With a recall of over 80% and query time of under 1 second, the algorithm is accurate and fast enough to observe data edited by a user and intelligently suggest a tope for validating and reformatting that data.

In the next two sections, we describe how we bring together prior work into the Toped^{++} editor. This prior work includes the topes model [14], an algorithm for inferring formats from examples [12], and an algorithm for generating a context-free grammar from a format [13]. In subsequent sections, we describe the new contributions in the current paper: algorithms for automatically creating reformatting rules and recommending topes. We then discuss empirical studies evaluating how well users can create and apply topes, as well as the scalability of the search-by-match algorithm.

THE TOPES DATA MODEL

Our approach models each kind of data as an abstraction called a "tope" [14]. Such an abstraction describes the formats, and the relationships among those formats, for one kind of data. For example, Figure 1 is a sketch of a tope describing our university's room numbers. Notice that each tope is a graph: the nodes are formats, and the edges are rules for reformatting values from one format to another.

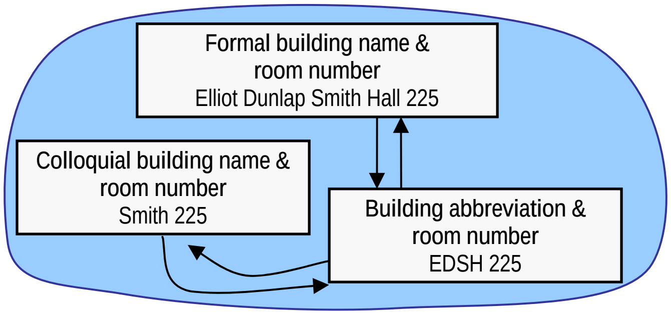


Figure 1. Notional depiction of a tope showing the three formats used for room numbers at Carnegie Mellon University

Each tope τ is a directed graph (F, T) that has a function associated with each vertex and edge. For each vertex f∈F, $\texttt{isa}_f$ is a fuzzy set membership function $\texttt{isa}_f$:∑*→[0,1], where ∑ is UniCode. For each edge (x,y)∈T, called a "transformation", $\texttt{trf}_{xy}$ is a function $\texttt{trf}_{xy}$:∑*→∑*.

For many kinds of data, it can be impossible to conclusively determine if a string is a valid instance. It is often useful to call out these "questionable" strings so that they can be double-checked by a person. To model this, each $\texttt{isa}_f$ defines a fuzzy set [15], such that $\texttt{isa}_f\texttt{(s)}$ indicates the degree to which `s` is an instance of the format. For example, user names in email addresses can be up to 64 characters, but anything over 32 characters is rare and deserves double-checking by a human. Thus, this tope's `isa` functions might return 1 if the user name is under 32 characters and 0.1 otherwise.

The `trf` functions reformat strings from one format to another. These functions can be chained at runtime, so although some topes could have a central format connected to each other format in a star-like structure, alternate topologies could be used in other cases, such as when a tope is conveniently expressed as a chain of sequentially refined formats.

THE TOPED^{++} EDITOR

To create a tope, a spreadsheet user highlights cells and clicks on a button in our Excel plug-in, which appears as a toolbar on-screen. Using an algorithm that we have described elsewhere [12], Toped^{++} infers a boilerplate tope implementation that describes most or all of the examples. Toped^{++} refers to a tope as a "data description" (Figure 2).

A data description has one or more variations, each of which is shown on a row of the screen. Each variation has one or more part, as well as separators between parts. A part can appear on more than one row; for example, the two variations in Figure 2 have the same two parts, but in different orders and with a different separator between parts.

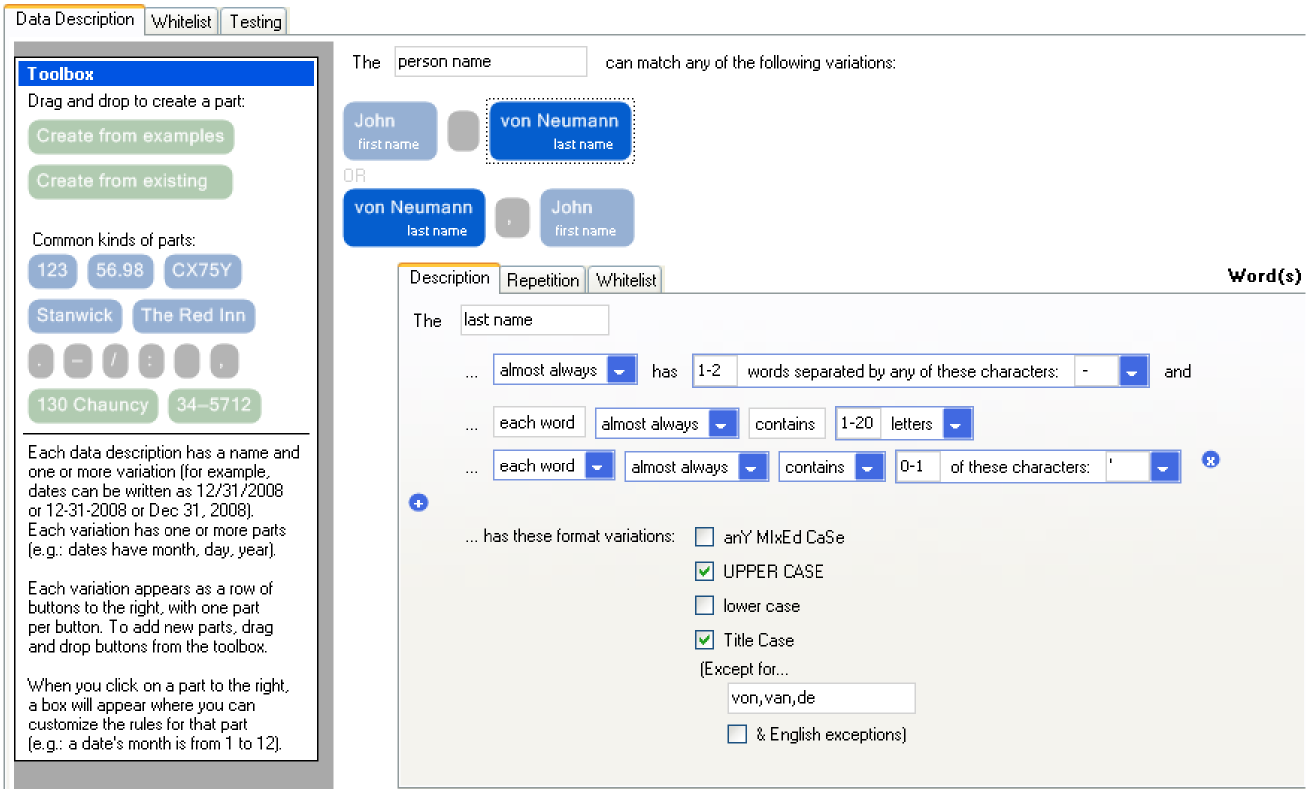
Each part has constraints, and each kind of part has different constraints. A Word-like part has constraints for specifying punctuation and other characters that may appear in instances of the part. A Numeric part has a numeric range and options for specifying the number of decimal digits allowed in instances of the part. A Hierarchical part matches another data description (recursively). Most constraints can be marked as "soft", in that a valid instance of the part might possibly violate the constraint (as in the example of a long email user name above). To edit a part's constraints, the user clicks on one of the part's icons (in any variation where the part appears), causing its constraints to appear in the part editor (on the screen's lower right).

To move parts around, the user can drag and drop part icons. To add another part to a variation, the user drags and drops an icon from the "toolbox" on the left side of the screen; newly created parts are "pre-loaded" with constraints that are usually appropriate for that kind of part. The editor supports the other operations typically supported by visual editors, such as copy/paste/delete.

Variations and formats

Each variation corresponds to one or more formats (nodes in the tope graph). Each variation may correspond to more than one format because each constituent part in a variation may also have variations of its own.

Specifically, each Word-like part has up to four variations, depending on the user's specification, and each Numeric part has one variation for each number of decimal digits specified as allowable by the user. For example, in Figure 2, the user has indicated that the "last name" has an upper case variation and a title case variation. The "first name" part has the same variations. Thus, the first variation in Figure 2 corresponds to four different formats (e.g.: "JOHN VON NEUMANN", "JOHN von Neumann", "John VON NEUMANN", and



Drag and drop to create a part:

Common kinds of parts:

Each data description has a name and one or more variation (for example, dates can be written as 12/31/2008 or 12-31-2008 or Dec 31, 2008). Each variation has one or more parts (e.g.: dates have month, day, year).

Each variation appears as a row of buttons to the right, with one part per button. To add new parts, drag and drop buttons from the toolbox.

When you click on a part to the right, a box will appear where you can customize the rules for that part (e.g.: a date's month is from 1 to 12).

The person name can match any of the following variations:

The last name

... almost always has 1-2 words separated by any of these characters: - and

... each word almost always contains 1-20 letters

... each word almost always contains 0-1 of these characters: '

... has these format variations:
□ anY MIxEd CaSe
☑ UPPER CASE
□ lower case
☑ Title Case
(Except for...
von,van,de
□ & English exceptions)

Figure 2. Editing a person name data description. Each variation appears on one row and has a series of parts that may be clicked to edit the part's constraints. A part may appear in multiple variations. Clicking ⊕ adds a constraint; clicking ⊗ deletes it.

"John von Neumann"). The second variation corresponds to four formats, as well.

A Hierarchical part can also have variations. Since a Hierarchical part references another data description, the part's variations are exactly the same as those of the referenced data description. For example, a user might have a spreadsheet cell containing a person's name and email address, as in "Brad Myers <bam@cs.cmu.edu>". A data description for this cell would need one variation, which would have a Hierarchical person name part, a " <" separator, a Hierarchical email part, and a trailing ">" separator. The person name part would reference the person name data description and thus have the 8 formats discussed above. The email address part would reference an email address data description, which could also have more than one format (e.g.: matching "bam@cs.cmu.edu" and "bam@CS.CMU.EDU"). If the email address had 2 formats, then the overall data description would have 8 x 2 = 16 formats.

In general, each format in the data description as a whole is uniquely distinguished by the following information:

- What variation does the format belong to? Membership in a variation indicates what parts and separators compose the format, as well as their ordering.
- Recursively, what is the format of each part in the format? This indicates how each part should be formatted prior to concatenation with other parts and separators.

Validating spreadsheet cells

After the user saves the data description, Toped^{++} implements each format's `isa` function by generating a context-free grammar [13]. The user-specified soft constraints from the part editor (Figure 2) are attached to the grammar's productions. Each `isa` returns 1 if the string matches the format's grammar, 0 if the parse fails entirely, and a value in between if the parse succeeds but violates soft constraints. The precise `isa` return value is determined by converting adverbs of frequency in soft constraints (e.g.: "often") to probabilities; there is surprisingly little variance in the probabilities that people assign to these words [10].

To detect cells that do not match a format selected by the user, each cell is tested against that format's `isa` function. If the return value is less than 1, then the cell is flagged with a red triangle and a tooltip containing an error message that is constructed by concatenating a list of all constraints violated by the parse. The user can also configure our Excel plug-in to only show messages for results that seem very likely to be incorrect (internally, those with `isa(s)<0.1`).

Comparison to the original tope editor prototype

As described above, Toped^{++} builds on three pieces of prior work: the topes model [14], an algorithm for inferring formats from examples [12], and an algorithm for generating a context-free grammar from a format [13]. We had previously combined these pieces of work to produce an early prototype that required users to edit each format individually [13], which was much more onerous than using Toped^{++}. For example, whereas Toped^{++} enables users to view and edit all 8 person name formats at once (Figure 2), users of our early prototype had to manually implement each format individually, which created a great deal of redundant work.

Moreover, using a separate editor, users then needed to implement trf functions for reformatting from one format to another. These reformatting rules were very brittle: if the user modified a format (perhaps by reordering parts), then the reformatting rules also had to be edited correspondingly. Toped^{++} saves users this effort, as we now describe.

AUTO-IMPLEMENTION OF REFORMATTING

Toped^{++} automatically implements trf functions, thereby eliminating the need for users to explicitly create trf functions in the first place or to update them after modifying formats. We describe the implementation of these functions beginning with individual Word-like and Numeric parts, then explain how these are combined into larger structures.

A single variation with a single Word-like part

If a tope has a single variation with a single Word-like part, then there is a one-to-one correspondence between the variations of the Word-like part (as determined by the checkboxes on Figure 2) and the formats of the data description. Upper/lower/title case formats match words that are in upper/lower/title case, respectively. The title case format also matches words in a user-specified exception list. Capitalization constraints are soft, so if an input at runtime does not match the required capitalization exactly, the format's isa function returns a value between 0 and 1.

As shown in Figure 2, the user can also select a Word-like format that allows words in any capitalization. This format contains a very weak soft constraint that is always false. That is, its isa returns 0.9998, which is high enough that no warnings appear at runtime (since, by default, tools are configured to display warnings only when isa returns less than 0.9). Yet it is less than 1 so that if a string also matches an upper/lower/title case format, then the input is interpreted as being an instance of that specialized format rather than an instance of the generic mixed case format.

For each pair of formats (x, y) in a tope with a single variation containing a single Word-like part, Toped^{++} automatically implements a function trf$_{xy}$ that reformats inputs into the capitalization required by format y. For example, if y is the upper case format, then each trf$_{xy}$ converts each character to upper case. If y is the title case format, then trf$_{xy}$ title-cases each word except for those in the exception list. (As an exception to the exception, English prepositions may be capitalized if they are the first word in the string, but explicit user-specified exceptions are left in the case specified by the user—such as “von” and “van” shown in Figure 2) If y is the mixed case format, then trf$_{xy}$ returns the input unaltered.

Thus, without any explicit effort by the user, Toped^{++} automatically implements reformatting functions to output strings matching the target format. If the user enables or disables a format (by toggling its checkbox in Figure 2), then Toped^{++} adds or removes trf functions to/from the tope accordingly.

A single variation with a single Numeric part

Numeric parts are handled similarly. In the part editor for Numeric parts, the user can specify that up to a certain number of digits d are allowed for that part, yielding d formats. For example, a part might be written as “4.560”, “4.56”, “4.6” or “5”. For each pair of formats (x, y) in a tope with a single variation containing a single Numeric part, Toped^{++} implements a trf$_{xy}$(s) that rounds s (or appends ‘0’ characters) so the output has the number of decimal digits required by y. As with a tope containing one Word-like part, the resulting format graph is totally connected by trf edges, so there is no need for successive reformatting: the input is rounded or ‘0’-padded once to give the output.

A single variation with a single hierarchical part

A hierarchical part matches some other data description t. Thus, if a tope u has a single variation with a single hierarchical part matching t, then u has one-to-one formats with t. For each pair of formats (x, y) in u, Toped^{++} implements a function trf$_{xy}$ that calls the corresponding trf in t. That is, the task of putting the part into some format y is recursively delegated to the reformatting functions in that part's tope.

Topes with more than one part or variation

The discussion above explained how Toped^{++} implements trf functions for single-variation, single-part topes. In general, topes have multiple variations and multiple parts.

Suppose that a tope has formats x and y. In particular, let y contain separators <s_0, s_1, …, s_m> and parts <y_1, y_2, …, y_m> in formats <f_1, f_2, …, f_m>. (Some separators might be blank; s_0 is to the left of the first part, and s_m is to the right of the last part.) A string parsed with x's grammar is reformatted into y as shown in Figure 3.

```
Start with an empty string str
For k = 0 through m
  If k > 0, and format x includes part $y_k$
    reformat the value of $y_k$ to $f_k$
    and concatenate it to str
  Concatenate $s_k$ to str
return str
```

Figure 3. Implementation of trf$_{xy}$

Intelligent systems often work well, but they can make errors. In our situation, some parts in y might also be parts in x, but others might not, which can cause errors that the system must detect and bring to the user's attention. For instance, in Figure 2, every format had the same parts (albeit in a different order). But there could also have been a variation (corresponding to several formats) with a middle name.

In that case, some formats would require parts that other formats lack. In cases like these, the output of $\mathtt{trf}_{\mathtt{xy}}$ might not match $\mathtt{y}$. To detect errors of this kind, the output of the transformation is always checked with $\mathtt{isa}_{\mathtt{y}}$, yielding an error message prompting the end user who provided the input to review and manually correct the output.

Note that the `trf` implementation in Figure 3 reformats each part $\mathtt{y}_{\mathtt{k}}$ to format $\mathtt{f}_{\mathtt{k}}$. This is achieved by treating $\mathtt{y}_{\mathtt{k}}$ as a single-variation, single-part tope, one of whose formats is $\mathtt{f}_{\mathtt{k}}$, and calling a corresponding `trf` that is implemented as explained in the sub-sections above. For example, to put a Word-like part into title case, Toped^{++} treats it as a single-variation tope with a single Word-like part and calls the `trf` function described above to put that part into title case.

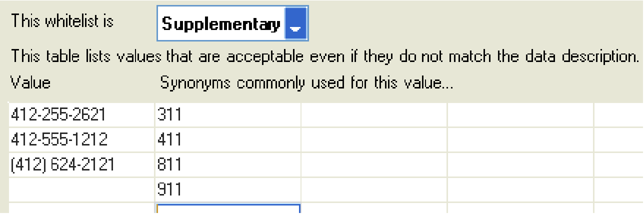
This approach for implementing reformatting rules yields a totally-connected graph in the tope, so the number of `trf` functions grows quadratically with the number of formats. However, the pseudo-code in Figure 3 is a twice-curried form of a generic function `trf(x,y,s)` that accepts a source format specification `x`, a destination format specification `y`, and a string `s`. It is this uncurried function that Toped^{++} actually generates from the data description to implement all the $\mathtt{trf}_{\mathtt{xy}}\mathtt{(s)}$ functions. At runtime, this function loops through the parts in `y` to compute $\mathtt{trf}_{\mathtt{xy}}\mathtt{(s)}$. In terms of runtime performance, the resulting function is several times slower than specialized functions would be, yet based on our experience, it is still much faster than human perception and thus adequate in an interactive setting.

WHITELISTS: LOOKUP TABLES FOR TOPES

These automatically-generated reformatting rules typically suffice for most data reformatting tasks, since the formats in most kinds of data are related via permutations of parts, changes of separators, changes of capitalization, and padding or rounding of numbers. However, for some kinds of data such as building names, it is desirable to create a fixed list of allowed values, each of which might have a synonym such as an abbreviation.

Sometimes, synonyms do not match any of the usual formats. For example, in Pittsburgh, "411" is a valid phone number equivalent to "(412) 555-1212", which provides directory assistance. To accommodate these situations, Toped^{++} supports whitelists. The "whitelist" tab of provides a grid where the user can enter a column of values that match the main data description, as well as synonyms (Figure 4).

Internally, Toped^{++} creates a new format `x` for each synonym column; $\mathtt{isa}_{\mathtt{x}}$ returns 1 for any string that matches a value in the column, or 0 otherwise. For each pair $\mathtt{(x_1,\ x_2)}$ of whitelist formats, Toped^{++} implements a $\mathtt{trf}_{\mathtt{x1x2}}\mathtt{(s)}$ that looks up the row of `s` in column $\mathtt{x}_1$, then returns the corresponding value in column $\mathtt{x}_2$. If this value is blank, then `s` is returned rather than a blank value; as with all reformatting, this result is checked with the target format's `isa`, which would return 0 in this case, resulting in a warning message.



This whitelist is Supplementary

This table lists values that are acceptable even if they do not match the data description.

Value	Synonyms commonly used for this value...
412-255-2621	311
412-555-1212	411
(412) 624-2121	811
	911

Figure 4. Editing a whitelist

For each non-whitelist format `y`, Toped^{++} reformats each value in the whitelist's leftmost column to format `y`, yielding an array $\mathtt{WL}_{\mathtt{y}}$. For each pair `(x, y)`, where `x` is a whitelist format and `y` is a non-whitelist format, Toped^{++} implements a $\mathtt{trf}_{\mathtt{xy}}$ that looks up the row of `s` in column `x` and returns the corresponding value in $\mathtt{WL}_{\mathtt{y}}$. (Again, if the value is blank, `s` is returned and checked against $\mathtt{isa}_{\mathtt{y}}$.) Conversely, `s` is reformatted from `y` to `x` by looking up the row of `s` in $\mathtt{WL}_{\mathtt{y}}$, then returning the corresponding value in `x`.

The user can configure the whitelist to be "strict" instead of "supplementary", so that only values in the whitelist are allowed. In this case, Toped^{++} attaches a constraint to the grammar for each non-whitelist format `y`, requiring that the input should be in $\mathtt{WL}_{\mathtt{y}}$. We have found whitelists to be an extremely succinct and direct way to create topes that match a fixed set of values, such as US states and building names.

AMBIGUITY

The preceding discussion explained how `trf` functions are created. However, to actually transform a string `s` to a format `y` selected by a user at runtime, it is necessary to first determine the *starting* format `x` of `s` so that the appropriate $\mathtt{trf}_{\mathtt{xy}}$ can be called. To achieve this, `s` is tested by each `isa` function, and the format `x` with the maximal `isa` score is chosen.

However, this raises the theoretical possibility of ambiguity: two or more formats might tie for the maximal `isa` score. In this case, arbitrarily choosing one format `x` as the starting format for reformatting could lead to errors. Specifically, there might be some other format `x′` for which $\mathtt{isa}_{\mathtt{x'}}\mathtt{(s)=isa}_{\mathtt{x}}\mathtt{(s)}$ but $\mathtt{trf}_{\mathtt{x'y}}\mathtt{(s)}\neq\mathtt{trf}_{\mathtt{xy}}\mathtt{(s)}$. It would be ideal to compute $\mathtt{trf}_{\mathtt{x'y}}\mathtt{(s)}$ and compare it to $\mathtt{trf}_{\mathtt{xy}}\mathtt{(s)}$ so that the user could be warned to check the output if the two possible outputs differ. However, this is practically infeasible, since there might be many such `x′` rather than just one.

Nonetheless, some `x′` can be ruled out, in the cases where $\mathtt{trf}_{\mathtt{x'y}}\mathtt{(s)}$ could not possibly differ from $\mathtt{trf}_{\mathtt{xy}}\mathtt{(s)}$. Specifically, ambiguity at the Word-like part and Numeric part levels can be disregarded. For example, regardless of whether `s` is interpreted as upper case or title case, its lower case equivalent is the same. Ambiguity between numeric formats is impossible by construction, since each Numeric part has a certain number of decimal digits.

Thus, ambiguity can only cause incorrect reformatting outputs if s matches one or more formats in a hierarchical part or in a tope as a whole. Even then, ambiguity will not affect outputs unless different formats x and x′ assign different values to parts. For example, the date "10/10/10" is ambiguous, but regardless of its interpretation, the correct reformatting to the international standard date notation (ISO 8601) is "2010-10-10". "03-10-08", on the other hand, is problematically ambiguous precisely because different formats match different substrings to month, day, and year.

Given all of these restrictions on when ambiguity can affect reformatting outputs, it turns out that ambiguity rarely causes incorrect reformatting outputs in practice. For example, none of the 900 randomly selected data values used in the experiment described below happened to have reformatting problems caused by ambiguity, and we only encountered a few such problems in a previous study of common spreadsheet data [14]. Perhaps the reason why ambiguity is not a problem in practice is that ambiguous notations are *generally* hard for humans to use (regardless of whether they are using a computer), so perhaps subconsciously or consciously, people avoid creating and tolerating notations that lead to ambiguous interpretations.

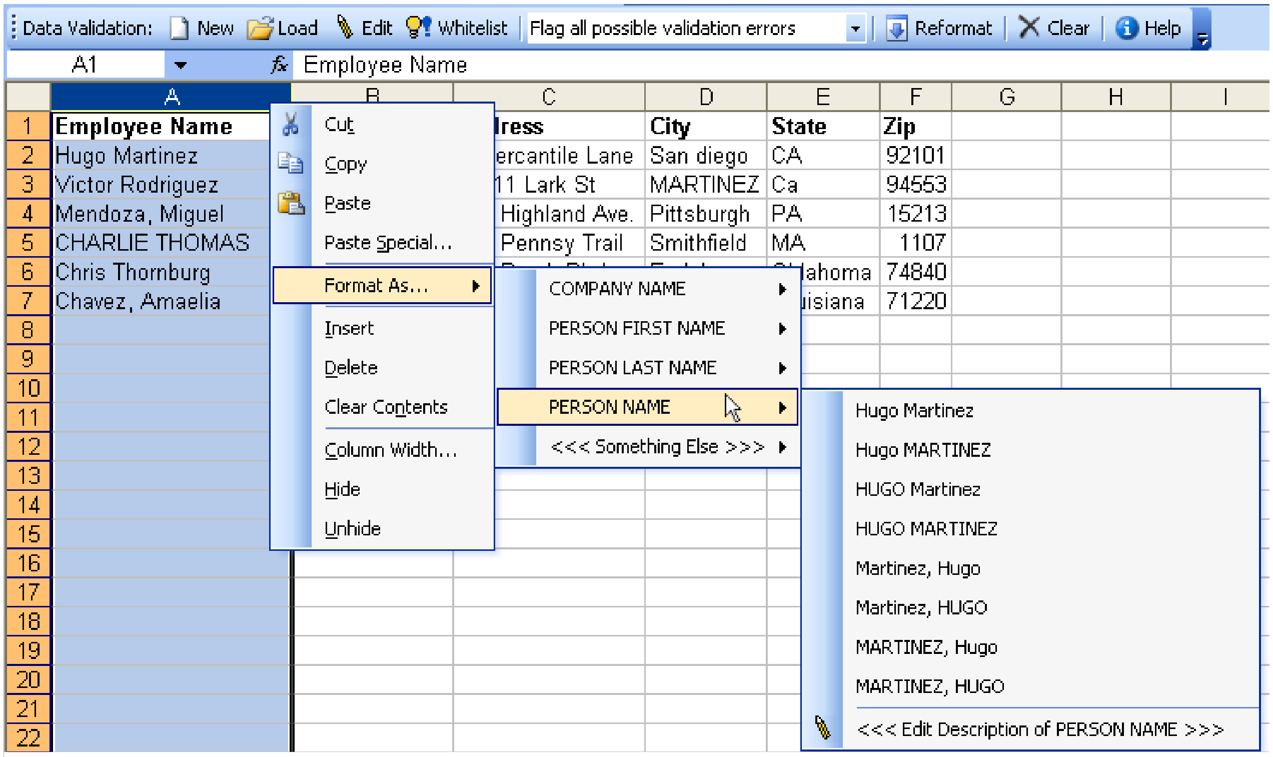
Nevertheless, for completeness, we may ultimately use the insights developed above to implement system enhancements that detect when different formats make different substring-part assignments. If this algorithm is fast enough for interactive use, then this check will be performed on every reformatted string, making it possible to warn users that specific outputs need inspection. Alternatively, we might use the algorithm to develop a testing feature to help users detect that the tope *could* yield reformatting errors due to ambiguity, as well as a feature to let users specify that a particular format should be preferred when its isa ties that of other formats.

RECOMMENDING TOPES AT RUNTIME

In a previous study, 32 topes accounted for 70% of all spreadsheet columns that we could categorize in a corpus of spreadsheets from the web [14]. In addition to these, a user might have dozens of uncommon or organization-specific topes that rarely appear in spreadsheets published on the web.

To reduce the need for users to browse through so many topes just to reuse one of them, our Excel plug-in shows a list of recommended topes in a context menu when the user right-clicks on cells (Figure 5). These recommended topes are identified based on the highlighted cells, as well as the words in the first row above these cells (since a column's first cell often has a header describing the column [1]).

For each recommended tope, the plug-in provides a submenu showing what the first non-header cell would become if reformatted to each format. If the user selects a format y from this submenu, then the highlighted cells are reformatted to y; if any cell's resulting value fails to match isa_y, then an error message in a tooltip appears on the cell. Alternatively, the user can add a new format to the tope by editing the tope or by adding highlighted cells to the whitelist.



	A	B	C	D	E	F	G	H	I
1	Employee Name		ress	City	State	Zip			
2	Hugo Martinez		ercantile Lane	San diego	CA	92101			
3	Victor Rodriguez		11 Lark St	MARTINEZ	Ca	94553			
4	Mendoza, Miguel		Highland Ave.	Pittsburgh	PA	15213			
5	CHARLIE THOMAS		Pennsy Trail	Smithfield	MA	1107			
6	Chris Thornburg				lahoma	74840			
7	Chavez, Amaelia				isiana	71220			

Figure 5. Browsing recommended topes and available formats for reformatting instances of one tope.

Basic recommendation algorithm

Our recommendation algorithm integrates traditional keyword-based search with search-by-match to find topes that match a set of example strings. Given a set of keywords K, a set of examples E, and a positive integer N indicating the maximum number of topes to return, the algorithm in Figure 6 retrieves the list of recommended topes T. This search-by-match algorithm first sorts topes according to how many keywords case-insensitively match words in tope names. The set of tope candidates is pruned so each remaining candidate matches at least a certain number of keywords; the threshold s_0 is chosen so at least N candidates remain under consideration. Each tope's keyword count is augmented with a score indicating how many examples appear in each tope's whitelist (in any column). Since $0 \leq w[t] \leq |E|$, whitelist matches effectively break ties between topes with the same keyword count. After another round of pruning, scores are incremented based on how well each tope matches the examples. Because $0 \leq c[t] \leq 1/(|E| + 2)$, isa scores effectively break ties between topes with the same keyword and whitelist counts.

```
Recommend(K, E, N)
 Let T = all topes

 For each τ∈T,
   s[τ] = |{k∈K : k is in τ's name}|
 Prune(T, s, N)

 For each τ∈T,
   w[τ] = |{e∈E : e is in τ's whitelist}|
   s[τ] = s[τ] + w[τ] / (|E| + 1)
 Prune(T, s, N)

 For each τ∈T,
   c[τ] = 1 / (|E| + 2)
   For each e∈E,
     cx = maximal isa(e) among τ's formats
     if (cx == 0) cx = 0.00001
     c[τ] = c[τ] * cx
   s[τ] = s[τ] + c[τ] / (|E| + 2)
 Prune(T, s, N)
 Sort T by s and return top N topes

Prune(T, s, N)
 Sort T by s
 Compute the maximal s0 such that
     at least N topes in T have s[t] ≥ s0
 For each tope t in T such that s[t] < s0
   Remove t from T
```

Figure 6. Pseudo-code for basic recommendation algorithm

Optimizations and CharSigs

The pseudo-code above is straightforward to understand, but it is not quite fast enough for interactive use, so we have optimized it by inverting the "For each" loops with the score computations. For example, rather than looping through all topes to count keyword occurrences, the algorithm uses an inverted index to retrieve topes that match each keyword. We use the same approach for whitelists.

Inverting the third loop to retrieve topes that match a certain example is more complicated. In contrast to retrieving topes by keyword, the potentially infinite number of values matched by each tope makes it impossible to index topes according to what examples they match. Instead, we index topes according to what strings they *might* match, based on topes' "character content".

For example, a phone number can never contain a dollar sign, so if an example string contains a dollar sign, then there is no need to test it against a phone number tope. Likewise, for a string like "ABC-DE20" that has 5 letters, 2 digits (for a total of 7 alphanumeric characters) and a hyphen, it is only necessary to test this string against topes whose instances could possibly contain those characters.

We use a new data structure called a CharSig to record the possible character content of a tope. A CharSig records the range of how many digits might occur, the range of how many letters might occur, the range of how many alphanumeric characters may occur, and the range of how many times each other character may occur. For example, if a phone number tope had two formats to match values like "800-555-1212" and "800.555.1212", its CharSig would be 10 digits, 10 alphanumerics, 0-2 hyphens, and 0-2 periods.

The CharSig is computed bottom-up. Each Numeric part's hyphen, period, and digit content is computed based on if the number could be negative, how many decimal digits (if any) are allowed, and how many digits are in the maximal and minimal allowed values. The content of each word in a Word-like part is read directly from its constraints. To compute the content of the entire Word-like part, content ranges are multiplied by the number of words allowed, then further adjusted upward based on separators between words. The character counts of parts and inter-part separators are then added together to yield the range of possible occurrences for each variation in the data description. As in the phone number example above, the data description's CharSig is computed by considering the disjunction of these variations.

At runtime, topes are selected from the CharSig index if they could contain the characters demonstrated by each example. For each character class cc (digits, letters, alphanumerics, and each other character) and each i in the range 0-10, the index records the set S^{i}_{cc} of topes that could have i instances of character class cc. It also records S^{∞}_{cc}, indicating topes that could have 11 or more instances. These sets are intersected to identify topes that could possibly match each user-specified example. For instance, to match the license plate number "ABC-DE20", it is only necessary to consider topes in $S^{5}_{\text{letter}} \cap S^{2}_{\text{digit}} \cap S^{7}_{\text{alnum}} \cap S^{1}_{-}$

These optimizations do not improve the asymptotic complexity of the algorithm, which remains $O(|T|)$. (Prune uses a bucket sort to achieve linearity, taking advantage of the fact that s[τ] can take on only a finite number of values, which ultimately results from the mapping from a finite number of distinct adverbs of frequency such as "often" in Toped^{++} to `isa` return values.) However, the simulation discussed below indicates that filtering topes by CharSig improves performance by a factor of 2, making it quite adequate for interactive use. When a new tope is created, inserting it into the index is essentially instantaneous, since computing the CharSig and updating the S^{i}_{cc} sets can be done in `O(1)` time with respect to the total index size.

We note in passing that a CharSig could also be computed bottom-up for regular expressions (regexps), suggesting that we may be able to use our tope recommendation algorithm to search regexps by example in future systems.

EVALUATING USABILITY

We conducted a within-subjects experiment to assess the usability of Toped^{++} for helping users to validate and reformat spreadsheet data. As a baseline, we compared Toped^{++} to how long it would take each user to perform the same tasks by manually typing changes into Excel.

Using emails and posters, we recruited 9 master's students, who were predominantly in the school of business. None had prior experience with Toped^{++}. To filter out users who already understood the concept of designing string recognizers, we screened participants to ensure that none had experience with regular expressions.

The experiment had four stages: a tutorial, three tasks with Toped^{++}, the same tasks but without Toped^{++}, and a user satisfaction questionnaire. As we did not counter-balance users (that is, giving some participants the manual tasks prior to the Toped^{++} tasks), users were more familiar with the goals when starting the manual tasks than when starting the Toped^{++} tasks. Consequently, this experiment probably measured *lower bounds* on the relative benefits of Toped^{++}. Users could take up to 30 minutes for the Toped^{++} tasks (though all participants finished sooner), but they could only spend 60 seconds on each manual task. We anticipated that this time limit would not weary participants with tedium but would still allow us to measure the average number of seconds need to manually validate and reformat a spreadsheet cell.

Task details

Each task required subjects to find and fix typos in one kind of data and to reformat all cells to a specific alternate format. For example, one task required participants to find typos in email addresses, some of which were missing a user name or ".com". Some cells had an extra email address. We asked participants to fix these typos or to flag them if they could not be fixed, and then to put the addresses into a format with the user name capitalized, as in "USERNAME@domain.com".

Different kinds of data were assigned to different participants. Three participants worked on person first names, person last names, and university names; three worked on course numbers, US state names, and country names; three worked on email addresses, phone numbers, and full person names. We organized the kinds of data into these groups because those in each group are similar: the first had single-word single-variation kinds of data, the second had data that could be reformatted using a whitelist, and the third had multi-part multi-variation data. We chose this organization because we did not anticipate that most users would be able to master all of the features of Toped^{++} in a 30 minute study.

For whitelist-based tasks, we provided participants with a table in Microsoft Word, which they could copy-and-paste into the Toped^{++} whitelist editor. We created these tables by copying HTML tables from the web; for example, the list of US states was copied off of Wikipedia. This saved participants the trouble of searching the web to find appropriate tables. However, this does not bias the study in favor of Toped^{++}, since participants would need to search the web anyway even if they performed the tasks manually (except, of course, for those rare users who have memorized all US state names, country names, or courses in a course catalog).

In terms of ecological validity, these kinds of data were among the most common kinds of data in the EUSES corpus of spreadsheets downloaded from the web [4][14]. For each kind of data, we randomly selected 100 strings from one spreadsheet column in the corpus.

Speed and accuracy

Using Toped^{++}, participants spent an average of 4.5 minutes on each task. Tasks in Group 2 (whitelist-validated data) took longer, at 6.9 minutes. Regardless of group, creating a tope required approximately 3-5 minutes, so the extra time for Group 2 tasks was not spent on creating topes, but rather on fixing typos identified by topes. Doing this was slow because it required referring to the whitelist (in Word) to find each erroneous value's correct spelling, then switching back to Excel to type the right value. Regardless of group, participants had a very low error rate, leaving an average of just under 0.1 fixable typo per column of 100 cells. (This is in addition to one column of cells for one kind of data, university names, where one participant put every cell into a different format than the format that we requested. However, he also did the same for the corresponding manual task, so we consider this a matter of misunderstanding instructions rather than a usability problem in the tools.)

Participants took different approaches to manually perform each task. In 14 cases, they first went down the column and fixed typos, then went back to the top and spent remaining time on reformatting cells. In 11 cases, they simply started at the top and reformatted cells one after the other; in this case, typos generally did not get fixed, as manual reformatting is fairly slow, so participants only edited a few cells and encountered few typos along the way. In the last 2 cases, participants were flummoxed about where to begin and gave up without changing any cells.

For each manual task, we computed the number of seconds spent fixing typos, the number of typos fixed, the number of seconds spent reformatting, and the number of cells reformatted. For each group, we computed the average time to fix each typo and average time to reformat each cell. (We omitted the two cases where users gave up.) Given the columns' rate of typos, we projected how long it would take users to manually fix every typo in each 100-cell column (Table 1).

Table 1. Comparing time required to fix typos and reformat 100-cell columns in each of three categories considered

	Minutes Required		Breakeven
	Toped^{++}	Manual	point
	(actual)	(projected)	(# cells)
Group 1: Single word data	3.0	5.0	60
Group 2: Whitelist data	6.9	15.9	43
Group 3: Multi-part data	3.6	10.2	35
Overall Average:	*4.5*	*9.6*	*47*

As when using Toped^{++}, finding typos in Group 2 data required repeatedly referring to the list in Word, which was relatively slow. Moreover, whereas Toped^{++} automated reformatting, saving the users from having to refer to the Word document during reformatting, manual reformatting required referring to Word, further slowing these tasks. For each group, we computed the number of cells that would need to be in the spreadsheet column to justify creating a tope. This breakeven point ranged from 35 to 60 cells. Thus, once a user has validated and reformatted a few dozen cells, the tope would have "paid for itself". Moreover, the tope could then be reused to validate and reformat an unlimited number of future spreadsheets.

User satisfaction

Our user satisfaction questionnaire asked each participant if he preferred using the new Excel feature or doing the tasks manually. Every person strongly preferred Toped^{++}. In addition, the questionnaire asked users to rate on 5-point Likert scale how hard Toped^{++} was to use, how much they trusted it, how pleasant it was to use, and if they would use it if it was integrated with spreadsheets or word processors. Every participant but one gave a score of 4 or 5 on every question (the good end of the scale). Moreover, two people verbally launched into detailed descriptions of how they wished a tool like this had been available in office environments where they previously had worked, in order to do reformatting.

EVALUATING TOPE RECOMMENDATION

To evaluate the responsiveness and accuracy of tope recommendation as the number of topes increases, we ran simulations using topes that we created with Toped^{++} for the 32 most common kinds of data that we previously found in the EUSES spreadsheet corpus [4][14]. Simulating what would happen if a user had a certain number of topes, we randomly chose a subset `T` of topes from these 32. For each tope, we randomly selected a corpus column that was known (based on our previous study) to have that that tope's kind of data. Using the words in the first cell as keywords `K`, and using examples `E` randomly selected from the other rows, we retrieved the recommended topes. We then checked if the correct tope appeared in the result set. We computed the average recall and query time for different values of `|T|`, `|E|`, and number of results `N`. We then repeated the process without keywords.

Results and discussion

The best recall resulted for `|E|=4` examples (Figure 7). Adding more examples caused the algorithm to spuriously match the whitelist of other topes. Adding more examples also substantially increased the time required to perform queries (Figure 8), suggesting that in practice, it is best to limit the number of examples passed to the recommender. Including keywords slightly improved recall. Recall rose substantially with `N`, so it is desirable to return multiple options rather than simply assuming that the first is best.

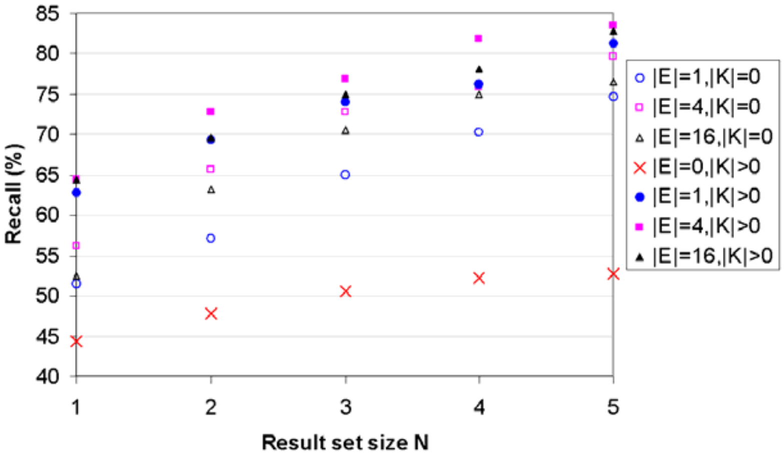


Figure 7. Keywords and |E|=4 examples gave maximal recall (|T|=32)

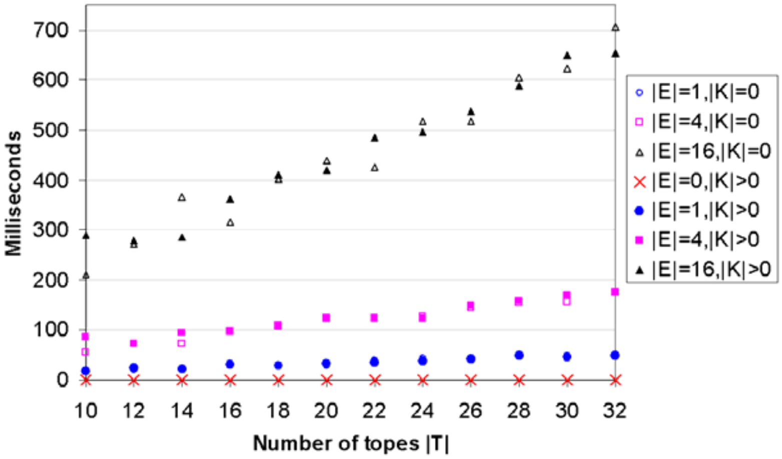


Figure 8. Recommendations (N=5) were computed in under 1 s.

A linear fit for the query time of the configuration `|E|=4 |K|>0` yields `time`=4.6 `|T|` + 25.9 ms (R^2=0.96). As we noted earlier, users are likely to have a few dozen topes on their computer. Populating the context menu with a list of topes selected from a collection of 50 topes would require approximately 250 ms. By instrumenting our code, we found that the CharSigs optimization was able to rule out approximately half of all topes during queries.

RELATED WORK

To our knowledge, our system is the first to integrate user-extensible string validation with executable reformatting rules. Regular expression editors like SWYN [2] and grammar editors like Grammex [7] enable users to specify patterns for matching data, but regular expressions and context-free grammars can only check data, not reformat it. Potluck [6] and Lapis [9] support simultaneous editing, in which a user types a change on one string, and the system automatically performs similar edits on other strings, but these edits are not stored as rules that can be later executed on other data. Nix's editing-by-example technique is similar to simultaneous editing, and it infers a macro-like program that records edits and can be replayed later. However, this technique does not support validation [11]. Microsoft Excel supports validation and reformatting of numeric data, but it offers no support for validating or reformatting strings.

RE-Trees support searching for regular expressions that match examples and, to our knowledge, are the only data structure that has been developed to support search-by-match [3]. Like our algorithm, the asymptotic complexity of RE-Tree search is `O(|T|)`, where here `|T|` indicates the number of regular expressions in the tree to be searched. Whereas inserting a tope into the CharSig index is `O(1)`, inserting a regular expression into an RE-Tree is `O(|T|)`.

The notion of topes as a model for kinds of data was influenced by information extraction research, where machine learning models are trained to identify and extract "named entities" such as company names and person names from raw text [8]. For example, the bigram "worked for" often follows a person name and precedes a company name. The absence of full sentences in spreadsheets and other structured datasets prevents using these algorithms for validation and reformatting of cells, but the notion of a named entity has strongly influenced our conception of topes [14].

CONCLUSION

In this paper, we have introduced an interface for users to specify formats, as well as an algorithm for automatically creating reformatting rules between formats. Our experiment shows that our system enables users to find typos and reformat spreadsheet cells much more quickly than is possible with manual editing, with extremely low error rates. By capturing the validation and reformatting rules in tope abstractions, and by providing an intelligent algorithm to recommend appropriate topes based on examples of the target data, we have made it possible to easily reuse reformatting rules on later spreadsheets. Based on these results, and users' enthusiastic responses to our satisfaction questionnaire, we believe that users would eagerly adopt Toped^{++} if it were made widely available.

ACKNOWLEDGMENTS

This work was funded in part by the EUSES Consortium via NSF (ITR-0325273) and by NSF under Grants CCF-0438929, CCF-0613823 and CCF-0811610. Opinions, findings, conclusions or recommendations expressed in this material are the authors' and not necessarily the sponsors'.

REFERENCES

1. Abraham, R., and Erwig, M. Header and Unit Inference for Spreadsheets through Spatial Analyses. *2004 IEEE Symp. Visual Lang. and Human Centric Computing*, 2004, 165-172.
2. Blackwell, A. SWYN: A Visual Representation for Regular Expressions. *Your Wish Is My Command: Programming by Example*, Morgan Kaufmann, 2001, 245-270.
3. Chan, C., Garofalakis, M., and Rastogi, R. RE-Tree: An Efficient Index Structure for Regular Expressions. *Intl. J. Very Large Data Bases, 12*, No. 2, 2003, 102-119.
4. Fisher II, M., and Rothermel, G. The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms. Tech. Rpt. 04-12-03, Univ. Nebraska-Lincoln, 2004.
5. Hall, J. A Risk and Control-Oriented Study of the Practices of Spreadsheet Application Developers. *Proc. 29th Hawaii Intl. Conf. System Sciences*, 1996, 364-373.
6. Huynh, D., Miller, R., and Karger, D. Potluck: Data Mash-Up Tool for Casual Users. *Lecture Notes in Computer Science, 4825*, 2007, 239-252.
7. Lieberman, H., Nardi, B., and Wright, D. Training Agents to Recognize Text by Example, *J. Auton. Agents and Multi-Agent Systems, 4*, No. 1, 2001, 79-92.
8. Marsh, E., and Perzanowski, D. MUC-7 Evaluation of IE Technology: Overview of Results. 7th Message Understanding Conf., 2001.
9. Miller, R., and Myers, B. Outlier Finding: Focusing User Attention on Possible Errors. *Proc. 14th Symp. User Interface Software and Technology*, 2001, 81-90.
10. Mosteller, F., and Youtz, C. Quantifying Probabilistic Expressions. *Statistical Science, 5*, 1, 1990, 2-12.
11. Nix, R. Editing By Example. *ACM Transactions on Program. Lang. Syst., 7*, No. 4, 1985, 600-621.
12. Scaffidi, C. Unsupervised Inference of Data Formats in Human-Readable Notation. *Proc. 9th Intl. Conf. Enterprise Information Systems*, 2007, 236-241.
13. Scaffidi, C., Myers, B., and Shaw, M. Fast, Accurate Creation of Data Validation Formats by End-User Developers. Submitted to *2nd Intl. Symp. End-User Development.*
14. Scaffidi, C., Myers, B., and Shaw, M. Topes: Reusable Abstractions for Validating Data, *30th Intl. Conf. on Software Engineering*, 1-10.
15. Zadeh, L. *Fuzzy Logic*. Tech Rpt. CSLI-88-116, Stanford University, 1988.