2nd International Symposium on End-User Development (ISEUD 2009), March 2009, to appear.

Fast, Accurate Creation of Data Validation Formats by End-User Developers

Chris Scaffidi, Brad Myers, Mary Shaw

Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15217
{cscaffid, bam, mary.shaw}@cs.cmu.edu

Abstract. Inputs to web forms often contain typos or other errors. However, existing web form design tools require end-user developers to write regular expressions ("regexps") or even scripts to validate inputs, which is slow and error-prone because of the poor match between common data types and the regexp notation. We present a new technique enabling end-user developers to describe data as a series of constrained parts, and we have incorporated our technique into a prototype tool. Using this tool, end-user developers can create validation code more quickly and accurately than with existing techniques, finding 90% of invalid inputs in a lab study. This study and our evaluation of the technique's generality have motivated several tool improvements, which we have implemented and now evaluate using the Cognitive Dimensions framework.

Keywords: Data validation, web macros, web applications.

1 Introduction

The success of Web 2.0 hinges on enabling end-user developers to create programs that collect information via the web. For example, accountants and financial analysts might create web macro scripts to automatically "screen scrape" data from web pages into spreadsheets [6][15], and marketing specialists or even administrative assistants might create web applications to receive information directly from people via web forms. This wide range of different users can then write programs to use the collected information for computation, report-generation, or generating new web pages.

Inputs often contain typos and other errors, so values should be validated when they are first received in order to prevent invalid data from jeopardizing the program's purpose. For example, if an end-user developer creates a web macro that scrapes data from a certain place on a web page, and the web page is later modified by the site owner, then the web macro may begin to read incorrect information and malfunction as a result [6][7]. Even when information is not collected automatically, inputs can still contain errors. For instance, prior studies have shown that people sometimes type the wrong kind of data into web form fields such as entering "12 Years old" into a street address field [16], which limits the usefulness of this information for generating maps, mining data, or creating reports.

However, existing tools for designing web macros and web forms can validate only a limited set of input types, and they present high barriers to specifying custom validation. For instance, end-user developers can create web forms with Microsoft Visual Studio.NET (which comes in an "Express Edition" for hobbyists, students, and end-user developers), and they can specify that textfield inputs should be validated against certain regular expressions (regexps), such as email addresses and zip codes. However, the list of available regexps is extremely short compared to the range of actual input fields that appear in real web applications. Examples of data types that lack pre-packaged regexps include city names and company names as well as organization-specific data types such as project numbers. Creating custom regexps or writing intricate JavaScript is beyond the abilities of many administrative assistants, accountants, and other end-user developers. Spreadsheets and other end-user development tools share these limitations.

Solving this problem is not as simple as providing an online repository of regexps that end-user developers could copy/paste into programming tools. Such repositories already exist, but they do not completely meet the users' needs, mainly because it is so difficult to obtain specialized regexps needed for particular applications. For example, the forum for the largest existing online repository[1] includes postings from people who want regexps that:

- Only accept a password if it has "at least 7 characters, at leaset [sic] 1 number, 1 lower case, 1 upper case and no spaces"
- Only accept a URL if its domain is not in a set of certain disallowed domains
- Only accept a zip code if the first three digits fall into a certain range

When people fail to find what they need in these repositories, the main problem is with the underlying regexp notation rather than with the repository per se, as it is extremely cumbersome and sometimes impossible to write regexps with the characteristics of the examples above (complex character counting, compound negative disjunctions, and numeric ranges).

To address the underlying need for a better notation, this paper presents a new interaction technique based on describing inputs as a series of constrained parts, resembling the way that end-user developers actually describe data. This reduces the time required to implement custom validation, and the automatically generated validation code displays targeted human-readable error messages to help application users fix invalid inputs. Our technique is integrated with programming-by-example and direct manipulation in a prototype editor called Toped. Given examples of the data to validate, Toped infers a format describing that data. It presents the format to the end-user developer, who can iteratively review, test and edit the format before using it to validate data in spreadsheets, web applications, web macros, and other programs.

Previous work described how to algorithmically infer formats from examples [17], how to use Toped formats to validate data in web macro editors and other programming platforms [6][18], and how to formally model collections of formats [16]. While this work briefly mentions Toped, it does not describe Toped's internal details, nor does it evaluate Toped's expressiveness and usability.

[1] http://regexlib.com/

In the current paper, we describe our format editor in detail as well as three studies: a formative interview-based formative study that originally led to our editor's design, an empirical evaluation of the editor's expressiveness and generality, and a user experiment evaluating its usability. Finally, we briefly summarize improvements in our most recent implementation of the editor, which we evaluate using the Cognitive Dimensions framework [5]. The contributions of this paper are:

- An interaction technique for describing formats as a sequence of constrained parts, which is adequate for representing a wide range of short, human-readable strings.
- A prototype tool that enables end-user developers to quickly create accurate formats, as well as editor innovations that will likely further enhance usability.

We review related work in Section 2. In Section 3, we describe the formative study that guided the design of the format editor, which we describe in Section 4. We evaluate the expressiveness and usability of the editor in Sections 5 and 6. These evaluations prompted improvements to our editor, which we describe and evaluate in Section 7. Finally, in Section 8, we conclude with a discussion of future work.

2 Related Work

To help end-user developers overcome some difficulties of the regexp notation, SWYN infers a regexp from example strings and presents it using a visual language for review and editing [1]. This language replaces regexps' exotic characters with shapes (for example, representing the Kleene operator * as a visual stack of letters and the disjunction operator | as a colored circle containing a set of options). SWYN adds simple negation to the regexp notation, using red shading to express substrings that are not allowed. Grammex [8] and Apple data detectors [13] describe strings as character sequences with context-free grammars (CFGs) rather than regexps. No usability studies have been done on Grammex or Apple data detectors, but we expect that the relative complexity of CFGs versus regexps make them less usable than SWYN. Unlike Toped, regexps and CFGs are ultimately expressed in terms of character sequences, which forces users to figure out how to translate high-level constraints into character sequence patterns. This can be difficult or even impossible, as in the case of examples mentioned in Section 1 (e.g.: a password if it has "at least 7 characters, at leaset [sic] 1 number, 1 lower case, 1 upper case and no spaces"). While Toped generates CFGs internally, users never see the grammars and can instead describe data as a series of constrained parts.

Lapis infers a pattern in a specialized textual language that end-user developers can edit and use to find outliers that do not match the pattern [9]. Whereas regexps, SWYN, and CFGs require end-user developers to describe a string as a character sequence, Lapis allows end-user developers to describe a string as a sequence of parts, each of which matches a regexp, literal string, or a certain primitive refined by "starts with", "contains", or "ends with" constraints. Toped eschews regexp-based constraints and instead offers a broad range of human-readable constraints that can be combined to validate parts of strings.

Toped also significantly differs from regexps, CFGs, and Lapis in that it supports soft constraints that need not always be true. This makes it possible to flag inputs that are questionable, but which should be accepted if they are double-checked.

Several tools check constraints over numeric data in particular programming platforms, rather than a format over string-like data. For example, Cues infers constraints over web service data [14], and Forms/3 infers numeric constraints over spreadsheet cells [2]. From a conceptual standpoint, Toped generalizes these number-oriented systems to include string-like data.

3 Formative Study

To learn how end-user developers describe data, we asked four administrative assistants to verbally describe two types of data (American mailing addresses and university project numbers) so a hypothetical foreign undergraduate could find those data on a hard drive. (We used this syntax-neutral phrasing to avoid biasing participants toward or away from regexps or any other particular notation.)

Participants described data as a series of named parts, such as city, state, and zip code. They did not explicitly provide descriptions of those parts without prompting, assuming that simply referring to those parts would enable the hypothetical undergraduate to find the data. When prompted to describe the parts of data, participants hierarchically described parts in terms of other named parts (such as an address line being a number, street name, and street type) until sub-parts became small enough that participants lacked names for them.

At that point, participants used constraints to define sub-parts, such as specifying that the street type usually is “Blvd” or “St”. They often used adverbs of frequency such as “usually” or “sometimes”, meaning that valid data occasionally violate these constraints. This use of not-always-true constraints stands in stark contrast to regexp-based validation, which classifies inputs as valid or invalid, with no shades of gray.

Finally, participants mentioned the punctuation separating named parts, such as the periods that punctuate parts of our university’s project numbers. Such punctuation is common in data encountered on a daily basis by end-user developers, such as hyphens and slashes in dates, and parentheses around area codes in American phone numbers.

4 Toped Format Editor

Based on the formative study results, we have designed a new tool for end-user developers to describe the format that inputs should match (Fig. 1). After a developer creates a format, its specification is stored and later used to check inputs. The process is conceptually similar to creating a regexp and using it to check inputs, except that Toped describes data as a sequence of named parts with constraints (Table 1). Toped is forms-based, a style of user interface known to help reduce memory load and errors, though at the risk of limited expressiveness and generality [12].

Step 1: Tell what kind of data your format is for... Mailing Address
Give your format a descriptive name... NNN Street Name Type.

Step 2: Describe the parts that make up each Mailing Address ...

+ part You can start from an example: Ok

Each Mailing Address has a part called the street number

The street number always has 1-5 of the following characters:
lowercase letters uppercase letters digits other characters:

+ info

+ part

Each Mailing Address has a part called the street name

The street name always has 3-10 of the following characters:
lowercase letters uppercase letters digits other characters:

The street name always is preceded by § and followed by
(You can leave one of these two fields blank if it does not apply.)

The street name can repeat 1-3 times, separated by §
The last separator in any list of repetitions is also §

+ info

+ part

Each Mailing Address has a part called the street type

The street type always has 2-9 of the following characters:
lowercase letters uppercase letters digits other characters: .

The street type always is preceded by § and followed by
(You can leave one of these two fields blank if it does not apply.)

+ info

+ part

Step 3: Test your format... Test Now
When you click the Test Now button, your format will be tested.

If you like, you can specify several Mailing Address examples in the left column of this spreadsheet ⟶
When you click Test, these examples will be checked to see if they match the format.

Copy All Paste All

15211 4th St	Ok
501 Highland Ave.	Ok
433 Amazon River Trail	Ok
767 Burgh Boulevard	Ok
#12 Locomotive Terr.	Does not match format

The street number always has 1-5 digits

Step 4: Save your format... Save Now

Fig. 1. Editing a format in Toped, after providing example street addresses and giving human-readable names to the inferred format and its three parts. Adding, removing, and reordering parts/constraints is accomplished with the + , ☒ , and ⏶ buttons. Each § indicates a space.

Table 1. Constraints that can be applied to parts.

Constraint	Description
Pattern	Specifies the characters in a part and how many of them may appear
Literal	Specifies that the part equals one of a certain set of options
Numeric	Specifies a numeric inequality or equality
Substring	Specifies that the part starts/ends with some literal or number of certain characters
Wrapped	Specifies that the part is preceded and/or followed by a certain string
Optionality	Specifies that the part may be missing
Repeat	Specifies that the part may repeat, with possible separators between repetitions
Reference	Specifies that the part matches another format

We named the editor “Toped”, after the Greek word “tope” for “place”, because each format validates a kind of data that has a natural “place” in the problem domain. That is, problem domains involve email addresses and salaries, rather than strings and floats, and it is the problem domain that governs whether a string is valid.

Our editor has four sections, as shown in Fig. 1. First, the user names the data to validate, such as “phone number” or “person name”, and names the format. It is important to distinguish between the name of the data and the name of the format, since many kinds of data can appear in more than one format (such as “08/16/2008” and “Aug 16, 2008” formats for US dates). Our system includes a format browser so the user can browse for formats using these names. Prompting the user to name the data also enables the editor to refer to the data by name in the next three steps.

Second, the end-user developer defines the format’s parts. For example, a US phone number has three parts: area code, exchange, and local number. The end-user developer can add, remove and reorder parts, and he can specify constraints on parts. Table 1 shows the available constraints, which we initially identified based on the formative study and later expanded as we used Toped to validate a variety of data.

In the third step of the editor, the end-user developer can enter example strings to test against the format. The editor displays a targeted error message in a mouse-over tooltip for each invalid test example. These targeted messages show a list of violated constraints. The end-user developer can iteratively debug the format.

Finally, the format is saved to a file, which can be reloaded and further edited.

4.1 Specific Editor Features

To help the end-user developer get started, Toped includes a textbox that accepts an example string. The editor identifies non-alphanumeric characters in the example and treats these as separators between parts. It then initializes a part in the editor for each part of the example and looks for a few basic constraints (e.g.: noticing that a word-like part starts with an uppercase letter). To provide further help, Toped can examine multiple examples of data and infer a single format describing the majority of those examples [17]. The examples are automatically copied into the testing feature (Step 3) to help the end-user developer review, test, and customize the format.

One problem with some editing tools is that spaces are invisible and (we suspect) hard to debug. To counter this, Toped makes spaces visible by representing them with a special symbol, §. Though this slightly reduces readability, it is preferable to having spaces that the end-user developer cannot see or debug. (In error messages, spaces appear as SPACE to avoid font-related problems.) To further improve readability and directness, Toped allows numeric ranges in textboxes that accept an integer. For example, users can specify that a part starts with “1”, “2-3”, or “4+” uppercase letters.

The Pattern, Literal, Numeric, Wrapped, and Substring constraints can be marked as “never”, “rarely”, “as often as not”, “more often than not”, “almost always”, or “always” true. We selected these adverbs of frequency because there is surprisingly little variance in the probabilities that people assign to these words, corresponding within a few percentage points to probabilities of 0%, 10%, 50%, 60%, 90%, and 100%, respectively [10]. This robust correspondence makes it feasible to integrate constraints with formal grammars (below). As users have trouble grasping mixtures of

conjunction and disjunction [11], all constraints are conjoined, and disjunction is expressed as two constraints with parallel sentence structure that are each not always true. Toped has an advanced mode, not shown in Fig. 1, for specifying conditional constraints that span multiple parts, such as the intricate leap year rules for dates.

4.2 Validating Data

Formats are not used to validate data directly. Instead, they are converted automatically into a grammar, which is then used at runtime to validate strings in web forms, web macros, and other end-user development platforms.

Generating a grammar and parsing strings. To create a grammar, Toped generates a hierarchy of CFG productions to represent Reference constraints, indicating that a part should match another format. It inserts separators between parts based on Wrapped constraints, and it inserts repetition productions based on Repeat constraints. Toped generates leaf productions based on Numeric, Pattern and Literal constraints, which indicate that certain characters or literal values appear in a part.

Like other CFGs, this basic CFG can only accept or reject strings. But the soft constraints supported by the Toped user interface are more expressive, since they can identify questionable inputs that might or might not be valid. Toped reflects this additional expressiveness in the grammar by augmenting the basic CFG with additional productions and constraints on productions. Each production constraint has a "penalty", which indicates that if a substring matches the production but violates the constraint, then the substring is questionable and might be invalid. This penalty is between 0 and 1, depending on the probability corresponding to the adverb of frequency on the constraint.

The Numeric, Pattern and Literal constraints can be directly attached to productions of respective parts. If a Wrapped constraint has an adverb of frequency, indicating that separators should occur but are not mandatory, then Toped augments the basic CFG with alternate productions where the separator does not occur, and it attaches a penalty to the alternate productions. Toped also generates alternate productions from Optionality constraints, but without penalties.

At runtime, strings are parsed according to the augmented grammar. Our parser is based on GLR [19], which runs in linear time (with respect to the length of the input string) when the grammar has low ambiguity, as is generally the case with grammars generated from Toped formats. We adapt GLR by incorporating constraints. When a production is completed, producing a variable v, GLR automatically treats this variable as valid and uses it to complete other productions waiting for v. In contrast, our adapted version of GLR associates a score with each variable instance (a parse tree node), ranging from 0 through 1. When a production p for a variable v is completed, the parser evaluates any constraints on p. It downgrades the score of v by multiplying the score by each violated constraint's penalty. If a variable instance with a downgraded score is later used on the right hand side of a production, then the parser uses this score to multiplicatively downgrade the score of the variable on the production's left-hand side. Thus, the score of each node in the parse tree depends on child nodes' scores.

In short, these multiplications use penalties from violated constraints to score each node, including the root. That way, when the tree is complete, the parser can return a score between 0 and 1 for the input as a whole. In cases of ambiguity, the parser selects the parse with the highest score.

Generating error messages. To generate error messages, the parser tracks a list of violated constraints and concatenates them into a message (e.g.: the tooltip in Step 3 of Fig. 1). When a parse totally fails (making it impossible to identify specific violated constraints), the parser generates a message by concatenating constraints associated with failed productions that should have been completed. The resulting message is targeted and descriptive, a significant improvement over typical hard-coded error messages such as "Please enter a valid phone number" on web sites, or "The formula you typed contains an error" in Excel.

Toped has been integrated with several end-user development tools, including Visual Studio.NET [18], the Robofox web macro tool [6], and Microsoft Excel [18]. Formats can be reused without modification in each of these programming platforms. While error messages are generated in the same way for each platform, they are displayed in different ways.

For example, when a format is associated with a web form field using Visual Studio.NET, targeted human-readable error messages appear alongside textfields when inputs are invalid (Fig. 2). If an input matches the CFG but violates soft constraints, the code shows the message so that the end user can correct the input, but the message is displayed in a popup window so that the user can override the warning. End-user developers can also specify alternate settings, such as showing overridable messages with an "Ignore" button alongside textfields rather than in a popup, or configuring the web form to always reject any input that violates soft constraints.

Phone:	911-555-1212	The area code never ends with 11
Address:	373Maple Dr.	The street number always has 1-5 digits The street name always is preceded by SPACE
City:	New haven	The city's word always starts with 1 uppercase letter
State:	CX	The state abbrev always is one of these: AL, AK, AZ...
Zip:	8445	The zip always has 5 digits

Fig. 2. Validating web form data.

A second programming platform, web macros, enables end-user developers to create scripts that scrape data from web sites, but scripts can sometimes read the wrong data from web pages if the pages are modified by site owners after the scripts are created. Toped formats help make Robofox web macro scripts more robust to web page changes. When constructing a script, an end-user developer can highlight a clipboard item, which is a variable that is initialized by a copy operation in the script. This opens Toped so the end-user developer can create a new format or select an existing format. Robofox then creates an assertion specifying that after the copy operation, the clipboard should contain a string that matches the format. At runtime, if a string violates any constraint in the format, then Robofox displays a warning popup to

explain that the assertion is violated, enabling the end-user developer to modify the script or cancel execution if necessary.

To validate Excel spreadsheets, end-user developers highlight some cells, click a button, and select a format file, which can be customized if desired. Alternatively, clicking another button causes Toped to infer a new format using the highlighted cells as examples, and then the new format is presented for editing. Based on the grammar generated from the format, the plug-in validates each highlighted cell and flags each invalid cell with a red triangle and a tooltip showing a targeted error message. End-user developers can browse through comments using Excel's Reviewing feature.

Toped's format editing user interface runs on Microsoft .NET 2.0 and is implemented in C#. Toped's parser for validating data against formats is available as a .NET library and as a Java library. All code has been made available as open source so that other researchers can continue to apply formats in new and innovative ways.

5 Expressiveness Evaluation

In many cases, "forms-based systems lack generality" [12] since they provide a relatively small set of primitives. Consequently, we evaluated the expressiveness of Toped by implementing formats for a range of data types commonly encountered by end users. As this was an evaluation of expressiveness, not usability, we implemented the formats ourselves as expert users.

5.1 Data and Method

To identify test data, we ran logging software for three weeks on four administrative assistants' computers. When a user filled out a web form in Internet Explorer (these users' preferred browser), the logger recorded the fields' HTML names and some text near each field (to capture the fields' human-readable labels).

For each field, the logger recorded a regexp describing the string that the user entered. (We recorded a regexp rather than the literal string to protect users' privacy.) To generate regexps, we converted each lowercase letter to the regexp [a-z], uppercase to [A-Z], and digits to [0-9], then concatenated regexps and coalesced repeated regexps (e.g.: user0@XYZ.EDU → [a-z]{4}[0-9]@[A-Z]{3}.[A-Z]{3}).

We manually examined many of the 5897 logged regexps and wrote scripts to gather fields into semantic families such as "email" and "currency" based on HTML names and human-readable text near the fields. As shown in Table 2, 5527 (93.7%) fell into one of 19 semantic families. Using the regexps as a reference, we created formats for the 14 asterisked families. We omitted 3 families (justification, description, and posting title) because each would have simply required a sequence of any characters. That is, it is doubtful that these fields have any semantics aside from "text." We omitted usernames and passwords because we wanted to post formats online and did not want to reveal formats of our users' authentication credentials. Finally, we used our parser to test formats with sample strings, which we generated by referring to concrete regexps in the log and by using our knowledge of formats' semantics (such as what might constitute an email address).

Table 2. Semantic families gathered from user data. Asterisked groups were used for testing.

Family name	Strings	Example regexp from logs	Formats Needed	Strings Not Covered
project number *	821	[0-9]{5}	1	1
justification	820	*Very long*		
expense type *	738	[0-9]{5}	1	
award number *	707	[0-9]{7}	1	
task number *	610	[0-9][A-Z]	2	
currency *	489	[0-9]\.[0-9]{2}	2	6
date *	450	[0-9]{2}\/[0-9]{2}\/[0-9]{4}	2	2
sites *	194	[a-z]{3}	1	
password	155	*Several characters*		
username	121	*Several characters*		
description	96	*Very long*		
posting title	65	*Very long*		
email address *	50	[a-z]{8}@[a-z]{7}\.[a-z]{3}	2	7
person name *	48	[A-Z][a-z]{5}\s[A-Z][a-z]{8}	2	
cost center *	41	[0-9]{6}	1	
expense type *	41	[0-9]{5}	1	6
address line *	37	[0-9]{3}\s[A-Z][a-z]{5} \s[A-Z][a-z]{4}	1	
zip code *	28	[0-9]{5}	1	
city *	16	[A-Z][a-z]{8}	1	

5.2 Results

We had little trouble expressing families as formats. Some required 2 formats, but this was reasonable, as web forms generally require inputs to be in a certain format. For example, we needed a format for dates like "10/12/2004" and another for dates like "12-Oct-2004".

When testing formats, we found that we had made 4 errors. Three were cases where we failed to mark a part as optional. The fourth error was an apparent slip of the mouse, in which we indicated that a constraint was often true rather than always true. The version of the editor that we used for this evaluation did not have the testing feature (Step 3 in Fig. 1). We noted that these errors probably could have been found if we had been able to test our formats when we created them. Therefore, after our evaluation, we added the editor's testing feature.

After correcting these errors, our formats covered 99.5% of the 4250 strings used for testing. The 22 strings not covered included 17 apparent typos in the original data and 4 cases that probably were not typos by the users (that is, they were intentionally typed), but we suspect that they may have been invalid inputs, nonetheless. For example, in 2 cases, users entered a month and a year rather than a full date. The final uncovered test value was a case where a street type had a trailing period, and the editor offered no way for us to express that a street type may contain a period but only in the last position, a limitation that has recently been addressed. Formats' effectiveness at identifying invalid values suggests that they are powerful enough for validating a variety of data types.

6 Usability Evaluation

We conducted a between-subjects experiment to assess Toped's usability. As a baseline, we compared Toped to Lapis (which was described in Section 2) because Lapis patterns are more expressive than regexps or CFGs, and were previously shown to be usable by end-user developers [9].

Using emails and posters, we recruited 7 administrative assistants and 10 graduate students, who were predominantly master's students in information technology and engineering. None had prior experience with Toped or Lapis, but many had some experience with programming or grammars. We paid $10 to each.

We randomly assigned each to a Toped or Lapis group. Each condition had four stages: a background questionnaire, a tutorial for the assigned tool, three validation tasks, and a final questionnaire.

The tutorial introduced the assigned tool, coaching subjects through a practice task and showing all tool features necessary for later tasks. Subjects could ask questions and work up to 30 minutes on the tutorial.

The validation tasks instructed subjects to use the assigned tool to validate three types of data. Subjects could spend a total of up to 20 minutes on these tasks and could not ask questions. Subjects could refer to the written tutorial as well as an extra reference packet extensively describing features of the assigned tool.

6.1 Task Details

In Lapis, text appears on the screen's left side, while the pattern editor appears on the right. End-user developers highlight example strings, and Lapis infers an editable pattern. Lapis highlights each string in purple if it matches the pattern or yellow if it does not. For comparability, we embedded Toped in a text viewer with the same screen layout and highlighting. Each example string on the left was highlighted in yellow if it violated any constraints or purple otherwise.

Each task presented 25 strings drawn from one spreadsheet column in the EUSES corpus, an existing collection of spreadsheets from the web [4]. Each column also contained at least 25 additional strings that we did not show but instead reserved for testing. All 50 strings were randomly selected.

The first task used American phone numbers, the second used street addresses (just the street address, not a city or state or zip), and the third used company names. We selected these types to exercise Toped on data ranging from highly structured to relatively unstructured. The data contained a mixture of valid and invalid strings. For example, most mailing addresses were of the form "1000 EVANS AVE.", but a few were not addresses, such as "12 MILES NORTH OF INDEPENDENCE".

We told subjects that the primary goal was to "find typos" by creating formats that properly highlighted valid strings in purple and invalid strings in yellow. To avoid biasing subjects, we did not use Toped or Lapis keywords in the description of validity. To further clarify the conditions for validity, the task instructions called out six strings for each data type as valid or invalid.

6.2 Results

We asked subjects to think aloud when something particularly good or bad occurred in the tool. One Toped subject interpreted these instructions differently than the other subjects, as she spoke aloud about virtually every mouse click. We discarded her data from analysis, leaving 8 subjects assigned to each group.

As shown in Table 3, we used the conservative Mann-Whitney (Wilcoxon) statistical test, since our sample was small and we could not assume normally distributed data (though all measures' medians were very close to the respective means).

Table 3. Results comparing Toped to Lapis.

	Toped	**Lapis**	**Relative Improvement**	**Significant? (Mann-Whitney)**
Tasks completed	2.79	1.75	60%	$p<0.01$
Typos identified				
On 75 visible strings	16.50	5.75	187%	$p<0.01$
On all 150 strings	31.25	9.50	229%	$p<0.01$
F1 accuracy measure				
On 75 visible strings	0.74	0.51	45%	No
On all 150 strings	0.68	0.46	48%	No
User satisfaction	3.78	3.06	24%	$p=0.02$

Tasks completed. In the allotted time, Toped subjects completed an average of 2.79 tasks, while Lapis subjects averaged 1.75 (Table 3), a significant difference (Mann-Whitney, $p<0.01$). Toped subjects were more successful at their primary goal, finding typos. Of the 18 actual invalid strings in the 75 visible strings, Toped subjects found an average of 16.5 invalid strings, compared to 5.75 for Lapis subjects, which was a significant difference (Mann-Whitney, $p<0.01$). In addition, of the 35 typos in the total set of 150 test strings, the completed Toped formats found an average of 31.25 invalid strings, whereas completed Lapis patterns found only 9.5, a significant difference (Mann-Whitney, $p<0.01$).

Accuracy. Finding invalid data is not sufficient alone. Validation should also classify valid data as valid. We evaluated accuracy using F1, a standard statistic commonly used to evaluate classifiers, with typical F1 scores in the range 0.7-1.0 [3]. F1 combines measures for "false negatives" and "false positives". Compared to simply counting classification errors, F1 more effectively "discourages classifiers that sacrifice one measure for another too drastically" [3]. The 23 completed Toped formats had an F1 of 0.74 on the 75 visible strings and 0.68 on all 150 strings, whereas the 14 completed Lapis patterns had respective scores of 0.51 and 0.46, though these inter-tool differences were not statistically significant (Mann-Whitney, $p<0.05$). Thus, Toped subjects completed more tasks without sacrificing accuracy.

User satisfaction. We assessed user satisfaction because end-user developers such as students and administrative assistants typically do not need to program to get their work done: they can choose a manual approach rather than a programmatic approach if they do not like their programming tool [12].

Subjects generally commented that Toped was easy to use, "interesting" and "a great idea". Most suggested other types of data to validate, such as email addresses, license plate numbers, bank record identifiers, and other application-specific data. One subject commented that it was unintuitive to represent "two options" (disjunction) as two constraints with parallel structure.

The satisfaction questionnaire asked subjects to rate on 5-point Likert scale how hard the tool 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. We found that we could combine answers into a moderately robust scale (Cronbach's alpha=0.74). On this scale, subjects reported an average satisfaction of 3.78 with Toped and 3.06 with Lapis, a significant difference (Mann-Whitney, p=0.02).

No confounds with background. Subjects had different job categories and varying experience with grammars and programming. Yet for each tool, we found no statistically significant effects (Mann-Whitney, $p<0.05$) on task completion, format accuracy, or user satisfaction based on this prior experience or job category.

6.3 Comparisons to other studies

Regexp study. Though not perfectly comparable, it appears that subjects completed our tasks with Toped more quickly and accurately than subjects completed tasks with regexps in a study during SWYN's development [1]. For each of 12 data types presented in random order, that study asked 39 graduate students to identify which of 5 strings matched a provided regexp that was written in one of four notations. Average speeds on the last six tasks (after subjects grew accustomed to the notations) ranged from 14 to 21 seconds per string, and error rates ranged from 27% to 47%. (No F1 was reported.) In contrast, Toped subjects were faster and more accurate, not only checking strings, but also constructing a format at an average of 15 seconds per string (373.8 sec / task) with a simple classification error rate of only 19%.

Formative study. As in our formative study, our initial questionnaire (prior to validation tasks) asked subjects to write descriptions of US postal addresses and person names. Slightly more than half of responses (17) were sentences calling out parts by name. In some cases, they specified constraints on parts, whether implicitly in names ("street *number*" implies the presence of digits) or explicitly ("5 digit zip code"). Constraints often included an adverb of frequency. This description style is consistent with the earlier results and a close match to Toped's style of interaction.

Our questionnaire also uncovered an additional way that people describe data as a series of constrained parts. Many questionnaire responses (15) were non-sentence templates listing parts by name and visually showing spaces or punctuation separating parts. As in the sentence-like descriptions, many constraints were implicit in names, but explicit constraints were rarely specified. Our earlier study probably did not un-

cover this visual way of describing a series of constrained parts because the formative study required participants to *verbally* describe data, whereas this questionnaire asked participants to *write* descriptions, and the written medium is much more conducive to communicating template-like (visual) descriptions.

7 Recent Editor Improvements

During the evaluations described above, we observed the need for several editor improvements, which we have implemented and will evaluate below.

7.1 Requirements for Improvements

Sharing parts between formats. During the expressiveness evaluation, we observed that implementing validation for a semantic family often required more than one format. In most cases, these formats had the same parts but different separators and different ordering of parts. For example, person names could be written as “Otto von Bismark” or as “von Bismark, Otto”. In addition to the multi-format semantic families shown in Table 2, we have observed many other kinds of multi-format strings such as credit card numbers (written as “1234 5678 9012 3456” or “1234567890123456” or “1234-5678-9012-3456”) and phone numbers. The parts in these formats have identical constraints—it is merely the manner of combining the parts that differs.

In such cases, it would be ideal if the user could create a part and reuse its constraints in multiple formats. In the existing editor, this would only have been feasible by putting the first name into a format of its own, the last name into a format of its own, and then referencing these formats when creating the person name formats. This would lead to a rapid increase in the number of formats and format management complexity. Perhaps a particular user might still want to store the first and last name formats separately, in order to validate fields that should only contain a first or last name rather than a full person name, but at least the end-user developer should be able to make that choice rather than having it forced onto him.

Referencing collections of formats. Another consequence of the multi-format nature of users’ data is that a format’s part might match one of several formats. For example, a month can be written as “Aug”, “August”, “08”, or “8”, and when a month is referenced in a date, humans typically can understand the date regardless of which format is used for the month (with caveats about ambiguity among the day, month and year parts, which we discuss later). The implication for our format editor is that it would be ideal if a user could not only reuse one format in another format, but if they also could reference a collection of formats when specifying a part in another format.

Tailoring constraints to kinds of parts. In our expressiveness study, we noted that several kinds of parts frequently appeared, and the kind of the part strongly influenced the applicable constraints. Numeric parts always contained numeric characters, never contained alphabetic characters, always had Numeric constraints, and rarely needed

Substring constraints. Word-like parts rarely contained numeric characters (with exceptions like usernames in email addresses), always contained alphabetic characters, sometimes contained punctuation, and often needed Substring constraints.

Despite the existence of these kinds of parts, Toped treated each part as a "generic" part, rather than tailoring the interface to each kind of part. Consequently, adding a new constraint to a part required selecting the desired constraint from the full list of available constraints, rather than a shortened list of constraints relevant to that kind of part. It would be ideal if the editor instead showed the most relevant constraints for each kind of part, with an option to add other kinds of rarely-relevant constraints.

Showing disjunction more clearly. Finally, during the user study, a single participant commented that it was unintuitive to represent "two options" (disjunction) as two constraints with parallel structure. Disjunction most commonly occurred when different separators could be used between parts (such as writing a North American phone numbers as "888-888-8888" or "(888) 888-8888"), or when the part could start or end with certain particular options. The disjunction of separator options is actually the same issue as having multiple formats with the same parts (discussed above). As for Substring disjunctions, it would be ideal if choices for starting or ending strings could be listed within the same constraint, so as to avoid multiple parallel constraints.

7.2 Toped+: An Improved Prototype for Editing Formats

Our latest format editor, called Toped$^+$, refers to a collection of formats as a "data description" (Fig. 3). The data description can contain one or more variations that contain parts interspersed with separators. This presentation was inspired by the 15 cases where usability study participants used non-sentence templates listing parts by name and visually showing spaces or punctuation separating parts. Toped$^+$ supports drag/drop and copy/paste operations for creating and manipulating parts, as well as instantiation of parts or an entire data description from examples.

Each part has constraints, which can be edited by clicking on the part. Toped$^+$ supports three kinds of parts—Numeric, Word-like, and Hierarchical (referencing another data description—and each icon in the Toolbox corresponds to a prototype instance of a part or separator. When the user drags a new part from the Toolbox on the left, the part's editor is "pre-loaded" with a default set of constraints that are usually appropriate for that kind of part. In addition, the user can add Pattern, Literal, Substring, and Repeat constraints to any kind of part. Substring constraints can contain a disjunction of options. To test part constraints, each part icon has a user-editable example, which is validated using the part's constraints (generating a targeted message in a tooltip if the example fails to meet the constraints).

Thus, this improved interface directly meets three of the four requirements above: sharing parts between formats, tailoring constraints to kinds of parts, and showing disjunction more clearly. Support for the fourth requirement, referencing collections of formats, is more complex and has implications that require deeper explanation.

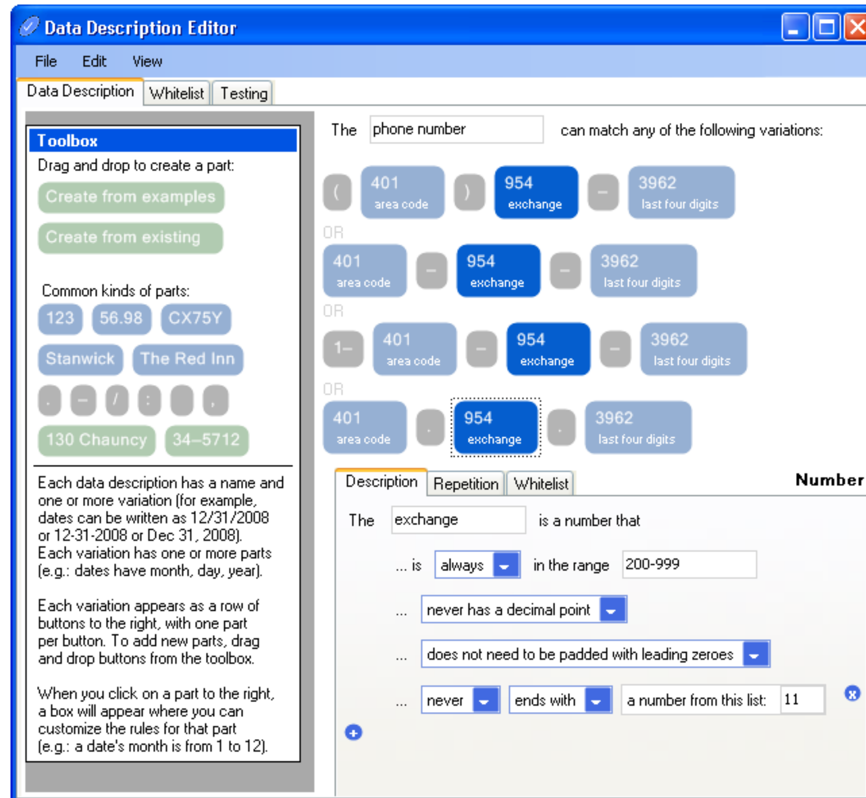

Fig. 3. Using Toped+ to edit variations of a phone number data description. Each variation appears on one row and consists of a series of parts (each of which may appear on multiple rows). Dragging and dropping a prototype's icon from the Toolbox creates a new part, and the editor also supports drag/drop re-arrangement of parts as well as copy/paste. Users can click the example in a part's icon to edit it, while clicking other parts of the icon displays widgets for editing its constraints, which are shared by every instance of the part. As in Toped, clicking adds a constraint, while clicking deletes the constraint. Users can toggle whether each space character is shown with a special character or as an invisible blank.

Referencing collections of formats. Toped+ allows users to create a Hierarchical part that references an existing data description, which may contain multiple variations. As a result, each variation actually corresponds to multiple formats. In this way, it is possible to quickly build up quite complex validation code.

For example, a month data description might have three one-part variations (the first of which would recognize "August", the second for "Aug", and the third for "8"), a day data description might have one variation, and a year data description might have two one-part variations (for two-digit and four-digit years). Concatenating hyphen separators with month, day and year parts (each referencing the respective data

description) would yield 6 formats visually represented on-screen by a single date variation. The user could duplicate this variation by copy/paste, then change the new variation's separators to slashes, yielding another six formats in just a few user interface operations. Dates are perhaps the most complex kind of string data that we have encountered, in terms of the number of formats. Yet because the parts are shared between variations, and because parts reference data descriptions rather than particular formats, it is possible to show many formats in relatively little space. This conciseness helps to greatly reduce the data description's visual complexity.

Ambiguity. A consequence of parsing against collections of formats, rather than particular formats, is the possibility of ambiguity. For example, should "02/06/08" be interpreted as a date in February, June, or August? Careful consideration, however, reveals that this is a false dilemma: the role of validation is to check whether this string matches *any* valid date, so it does not matter which date it refers to.

Forgetting about Toped for the moment, it would be perfectly reasonable for a skilled programmer to validate dates with a (very complex) regexp that had a disjunction of many different date formats, even though the regexp would be incapable of indicating which date format had been used. The regexp's job is to identify invalid dates, not to identify a valid date's format. On the other hand, if the programmer wanted users to enter values in a particular format, then he would instead use a more specific regexp that only recognized that particular format. Unfortunately, this would require the programmer to maintain a separate regexp for each particular format, since it would not be feasible to reuse the general-purpose format.

The same reasoning also applies to Toped+, but with improved reusability of validation code. If an end-user developer wants to validate dates against a specific format, then (unlike in the case of regexps), a general-purpose data description can still be reused. In this case, the user associates the data description with the input field and provides an *example* of the preferred format to our plug-in for Microsoft Excel or Visual Studio.NET. The plug-in calls our parser to parse the example in order to identify the desired format. (Obviously, the example must match one format, as in "12/31/99", in order to unambiguously specify the preferred format.) The plug-in associates this preferred format with the field so that at runtime, inputs are *checked against the demonstrated format* rather than the entire data description. Thus, Toped+ supports validating inputs against particular formats, but without the hassle of maintaining separate validation code for every format.

7.3 Evaluating Toped+ Improvements

Cognitive Dimensions is an established framework for qualitatively evaluating non-orthogonal aspects ("dimensions") of a notation, programming language or user interface, and for discussing trade-offs between different designs [5]. Example dimensions include Closeness of mapping, Abstraction gradient, Juxtaposability, and Visibility. Below, we consider 12 dimensions that are highly relevant to the task of creating formats, in order to identify strengths and weaknesses of Toped+ relative to Toped.

Closeness of mapping. Both Toped$^+$ and Toped have excellent closeness of mapping to the problem domain, since like end-user developers, they describe formats as a sequence of constrained parts. We believe that this closeness of mapping was a key ingredient to Toped's success in the user study. Moreover, Toped$^+$ constraints are associated with parts, rather than appearances of parts in particular formats, yielding better closeness of mapping to the kinds of data that appear in the real world.

Abstraction gradient and Hidden dependencies. Both Toped and Toped$^+$ require users to comprehend and manipulate constraint, part, and format abstractions. In addition, Toped$^+$ requires users to comprehend variations and data descriptions. Yet Toped$^+$ is not "abstraction-hungry" in the sense of requiring users to create multiple variations if they are not needed for a particular data description.

The usual problem introduced by reusable abstractions is the possibility of hidden dependencies. This caution applies to Toped$^+$, since if a user edits a part by clicking on its icon in one variation, then any changes will also affect the part as it appears in other variations. We have attempted to mitigate this potential for confusion by highlighting every one of a part's icons throughout the data description when a user clicks on a part (Fig. 3). Another form of hidden dependency, which applies to both Toped and Toped$^+$, is the fact that formats are indirectly affected when referenced formats change. Future versions of Toped$^+$ should mitigate this potential for confusion by listing the data descriptions that reference the data description currently being edited.

Juxtaposability, Diffuseness, and Visibility. In Toped, it was impossible to view two formats side-by-side at the same time. Toped$^+$ places formats adjacent to one another, making it straightforward to see differences in separators and part ordering.

Toped and Toped$^+$ both suffer from a common problem of visual languages in requiring a great deal of screen space. Toped$^+$ mitigates this diffuseness by only showing a part's constraints when the user clicks on the part's icon, which frees up enough screen space to show several variations at the same time. This design decision improves visibility but decreases the juxtaposability of parts' constraints, since it is therefore impossible to view two parts' constraints simultaneously. Fortunately, it is fairly rare that two parts need to have similar constraints, so we believe that it is more valuable to show multiple variations than multiple parts at the same time.

Error-proneness and Premature commitment. We expect the constraint editors in Toped$^+$ will help reduce error-proneness relative to Toped, since users would need to make a special effort to create senseless combinations of constraints.

The trade-off with this decision is that users must choose a particular kind of part before configuring the part's constraints. This is a form of premature commitment, since the user must manually back out of an error to a previous state (by deleting the part) before performing a more correct operation. It would be ideal if Toped$^+$ provided a feature to change the kind of a part, but we believe that it is infeasible to provide such a feature because of the richness of the supported constraints. Therefore, we have attempted to mitigate this problem by making it simple to delete a part (by clicking on it and typing Control+Delete, or selecting Delete under the Edit menu) and to create a part (by dragging a prototype from the Toolbox).

Viscosity. Toped+ greatly reduces the effort required to make changes to programs (viscosity). One reason is that simple operations are faster because Toped+ supports drag/drop and copy/paste. Another reason is that fewer user interface operations are required because it is so easy to reuse constraints and data descriptions.

Hard mental operations, Progressive evaluation, and Secondary notation. We have tried to minimize the need for hard mental operations in both Toped and Toped+ through support for inferring formats (or even particular parts in Toped+) from examples. Progressive evaluation is supported in Toped+ by letting the user enter an example (in the part's icon) that is validated with the part's constraints. In addition, the user can enter a list of examples to test the data description as a whole—just as a format could be tested in Toped. Moreover, unlike in Toped, the example strings entered for testing in Toped+ are actually stored with the data description, so they can serve as a form of "use case" secondary notation that provides additional information about the function of the data description. In both editors, users provide human-readable names for parts and data descriptions, which is a form of secondary notation.

8 Conclusion and Future Work

The central insight described in this paper is that end-user developers tend to describe string inputs as a series of constrained parts. We have found that this way of describing data is expressive enough for validating many kinds of data, and we have used this insight to design a tool that helps end-user developers create validation formats more quickly and accurately than is possible with existing tools. Our studies identified the need for editor improvements that we have implemented. Based on an analysis using the Cognitive Dimensions framework, we believe that the enhanced editor will be even more usable than the first version. We conclude that describing strings as a series of constrained parts is an effective approach for structuring validation code.

Future work will continue to develop Toped+ by providing highly-usable support for implementing functions that transform strings among the different formats of a data description. We have already prototyped a minimalist tool for implementing basic inter-format transformations, such as fixing capitalization and tweaking separators [16]. However, we have not integrated this tool with Toped+, since we anticipate that can do even better by developing algorithms that *automatically* implement these basic transformations based on the layout of a data description's variations. We will also support custom transformations with a general-purpose language such as JavaScript.

In addition, we are designing a repository that will enable end-user developers to publish, find, and reuse data descriptions. As noted in the introduction, the main problem with existing repositories is with the underlying notation, rather than the idea of a repository per se. Since Toped+ provides a means for end-user developers to inspect and modify formats in a notation that has strong closeness of mapping to the problem domain, we anticipate that a final summative user study will show that integrating a repository with Toped+ enables end-user developers to conveniently browse, select, reuse and customize data descriptions to validate data.

9 Acknowledgements

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

10 References

1. Blackwell, A.: SWYN: A Visual Representation for Regular Expressions. In: Your Wish Is My Command: Programming by Example, Morgan Kaufmann, pp. 245-270 (2001).
2. Burnett, M., et al.: End-User Software Engineering with Assertions in the Spreadsheet Paradigm. In: Proc. 25th Intl. Conf. on Software Engineering, pp. 93-103 (2003).
3. Chakrabarti, S.: Mining the Web: Discovering Knowledge from Hypertext Data, Morgan Kaufmann (2002).
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. of Nebraska (2004).
5. Green, T., and Petre, M.: Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. J. Visual Lang. and Computing, 7, pp. 131-174 (1996).
6. Koesnandar, A., et al: Using Assertions to Help End-User Programmers Create Dependable Web Macros. In: Proc. 16th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering, to appear (2008).
7. Lerman, K., Minton, S., and Knoblock, C. Wrapper Maintenance: A Machine Learning Approach. J. Artificial Intelligence Research, 18, pp. 149-181 (2003).
8. Lieberman, H., Nardi, B., and Wright, D.: Training Agents to Recognize Text by Example, J. Auton. Agents and Multi-Agent Systems, 4, 1, pp. 79-92 (2001).
9. Miller, R., and Myers, B.: Outlier Finding: Focusing Human Attention on Possible Errors. In: Proc. 14th Symp. on User Interface Software and Technology, pp. 81-90 (2001).
10. Mosteller, F., and Youtz, C.: Quantifying Probabilistic Expressions. Statistical Science, 5, 1, pp. 2-12 (1990).
11. Myers, B., Pane, J., and Ko, A.: Natural Programming Languages and Environments. Comm. ACM, 47, 9, pp. 47-52 (2004).
12. Nardi, B.: A Small Matter of Programming: Perspectives on End User Computing, MIT Press (1993).
13. Nardi, B., Miller, J., and Wright, D.: Collaborative, Programmable Intelligent Agents. Comm. ACM, 41, 3, pp. 96-104 (1998).
14. Raz, O., Koopman, P., and Shaw, M.: Semantic Anomaly Detection in Online Data Sources. In: Proc. 24th Intl. Conf. on Software Engineering, pp. 302-312 (2002).
15. Safonov, A. Web Macros By Example: Users Managing the WWW of Applications. In: CHI'99 Extended Abstracts on Human Factors in Computing Sys., pp. 71-72 (1999).
16. Scaffidi, C., Myers, B., and Shaw, M.: Topes: Reusable Abstractions for Validating Data, In: Proc. 30th Intl. Conf. on Software Engineering, pp. 1-10 (2008).
17. Scaffidi, C.: Unsupervised Inference of Data Formats in Human-Readable Notation. In: Proc. 9th Intl. Conf. on Enterprise Information Systems-HCI Volume, pp. 236-241 (2007).
18. Scaffidi, C., et al: Using Topes to Validate and Reformat Data in End-User Programming Tools. In: Proc. 4th Workshop on End-User Software Engineering, pp. 11-15 (2008).
19. Tomita, M.: An Efficient Augmented-Context-Free Parsing Algorithm. J. Computational Linguistics, 13, 1-2, pp. 31-46 (1987).