Topes: Enabling End-User Programmers to Validate and Reformat Data

Christopher Scaffidi

CMU-ISR-09-105
May 2009

School of Computer Science
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was funded in part by the EUSES Consortium via NSF (ITR-0325273) and by NSF under Grants CCF-0438929 and CCF-0613823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the sponsors.

Keywords: end-user software engineering, end-user programming, data validation, dependability, assertions, data reformatting, data formats, data consistency, web macros, web applications, spreadsheets, programming by demonstration.

Abstract

Millions of people rely on software for help with everyday tasks. For example, a teacher might create a spreadsheet to compute grades, and a human resources worker might create a web form to collect contact information from co-workers.

Yet, too often, software applications offer poor support for automating certain activities, which people must do manually. In particular, many tasks require validating and reformatting short human-readable strings drawn from categories such as company names and employee ID numbers. These string-containing categories have three traits that existing applications do not reflect. First, each category can be multi-format in that each of its instances can be written several different ways. Second, each category can include questionable values that are unusual yet still valid. During user tasks, such strings often are worthy of double-checking, as they are neither obviously valid nor obviously invalid. Third, each category is application-agnostic in that its rules for validating and reformatting strings are not specific to one software application—rather, its rules are agreed upon implicitly or explicitly by members of an organization or society.

For example, a web form might have a field for entering Carnegie Mellon office phone numbers like “8-3564” or “412-268-3564”. Current web form design tools offer no convenient way to create code for putting strings into a consistent format, nor do they help users create code to detect inputs that are unusual but maybe valid, such as “7-3564” (since our office phone numbers rarely start with “7”).

In order to help users with their tasks, this dissertation presents a new kind of abstraction called a “tope” and a supporting development environment. Each tope describes how to validate and reformat instances of a data category. Topes are sufficiently expressive for creating useful, accurate rules for validating and reformatting a wide range of data categories commonly encountered by end users. By creating and applying topes, end users can validate and reformat strings more quickly and effectively than they can with currently-practiced techniques. Tope implementations are reusable across applications and by different people, highlighting the leverage provided by end-user programming research aimed at developing new kinds of application-agnostic abstractions. The topes model demonstrates that such abstractions can be successful if they model a shallow level of semantics, thereby retaining usability without sacrificing usefulness for supporting users’ real-world goals.

Acknowledgements

Thank you to God for infusing the world with an unending wealth of interesting phenomena. Thank You for instilling in me a personality that gets satisfaction from learning about what You have made. In some ways, I feel like I didn't invent topes but rather found them by observing how the people in Your creation communicate with each other.

Thank you to Karen, my loving and beloved wife, for sharing the joys and trials of life with me. Thank you for sacrificing your time and convenience in order that I could work these years at Carnegie Mellon. I know that you pay a price for each day that I spend here, and I am grateful that you have never begrudged the cost.

Thank you to my parents for teaching me to work hard. Thank you to Mom for encouraging me, for as long as I can remember, to ask questions and to look for answers. Thank you to Dad for taking on the burden of not only raising five children but also taking on the joy of sharing life with us. When I think about the two of you, I often remember that the apple never falls far from the tree, and whatever accomplishments I have are an outgrowth of your love for me.

Thank you to Mary, my clever and kind advisor, for modeling a deep curiosity about life and the world. Thank you also for being my friend both inside and outside the office. It is no exaggeration to say that you are the best advisor that I have ever heard of or met.

Thank you to Brad, Jim, and Sebastian for going beyond the role of thesis committee and becoming partners in fruitful collaboration. Likewise, thank you to the members of the EUSES Consortium, particularly Allen and Margaret, for collaborating with me. Thank you especially to Brad, who has generously helped me in thousands of large and small ways. I appreciate the useful ideas and insights that all of you have provided.

Finally, thank you to the EUSES Consortium and to NSF for funding this research (via ITR-0325273, CCF-0438929, and CCF-0613823).

Table of Contents

Figures

Table 1. Preliminary work leading to this thesis

Chapter or Section	[100]	[101]	[102]	[103]	[104]	[105]	[106]	[107]	[108]	[109]	[110]	N/A
1 – Introduction												x
2.1 – End user counts											x	
2.2 – Info. worker survey					x							
2.3 – Info. worker contextual inquiry				x								
2.4 – Hurricane Katrina interviews										x		
2.5 – Asking users to describe data						x						
2.6 – Requirements for features												x
2.7 – Related work												x
3 – Topes model									x			
4.1 – TDE UI walk-through						x	x					
4.2 – TDE system overview								x				
4.3 – Toped^{++} details						x						
4.4 – Inferring data descriptions	x											
4.5 – Mapping variations to formats						x	x					
5.1 – TDE support for validation						x						
5.2.1 – Expressiveness: web form data						x						
5.2.2 – Expressiveness: spreadsheets									x			
5.2.3 – Usability for validation						x						
6.1 – TDE support for reformatting						x	x					
6.2.1 – Expressiveness: web form data									x			
6.2.2 – Usability for reformatting							x					
7.1.1 – Add-ins			x									x
7.1.2 – Search-by-match							x					
7.1.3 – Sharing data descriptions												x
7.2.1 – Topes' application-agnosticism									x			
7.2.2 – Scalability of search-by-match							x					
8.1 – Traits of reusable end-user code		x										
8.2 – Repository's advanced search												x
8.3 – Predicting reuse of web macros		x										
9 – Conclusion												x

Chapter 1. Introduction

Software has become an indispensible part of work and life. By 2012, 90 million people in America alone will use computers at work, while millions more will use computers at home. According to recent surveys, "86% of employed Americans use the internet or email at least occasionally" either at home or at work [66], 76% of online Americans have used the internet to purchase products online [66], and 11% of online American adults have even used Twitter [58]. While many people rely on software for casual tasks like communicating with family, software is also an essential part of productivity in businesses, universities, and governments. People rely on it for scheduling meetings, managing customer databases, collecting information from workers, and much more [66].

Many software applications actually allow users to create new custom programs to complete tasks more quickly or effectively. As an example of end-user programming, an accountant might create a spreadsheet to perform calculations so that he can complete a report more quickly than he could through performing the calculation by hand. An administrative assistant might create a web form in order to collect information from coworkers, thereby dispensing with hard-to-manage paper forms. To a lesser extent, end users also program at home as a way of capturing knowledge or expertise [96]. In end-user programming, the program is a tool—often disposable, but nonetheless a valuable tool for achieving real-world goals.

Yet applications often require end users to perform some tedious and error-prone activities that impede completing tasks. Many of these activities involve validating and reformatting data values. Existing applications provide no practical way for users to automate these validating and reformatting activities.

For example, during a contextual inquiry described later by this dissertation, I observed an office manager attempting to create an employee roster by copying data from a few intranet sites into a spreadsheet. This and many other user tasks involved categories of short human-readable data, such as phone numbers and job titles. While instances of these categories can all be represented as strings, not every string is a valid instance of each category. For example, "Student" is a valid string, but it is not a valid phone number. It could be a job title, though a questionable one because “Graduate Assistant” is actually the official job title at the user’s organization, Carnegie Mellon University. In this and other tasks observed during the contextual inquiry, users chose to accept and use questionable values, discard them, or check them against another source (such as an online directory). If the user had encountered an obviously invalid job title such as “10 Main Street”, he almost certainly would have not bothered to check it. In short, rather than categorizing strings as simply valid or invalid, participants treated the validity of data categories in a *non-binary* manner. This enables users to triage data and focus effort on only double-checking questionable strings that are neither obviously valid nor obviously invalid.

User tasks were further complicated by the fact that each instance of some categories can be written in multiple formats. For example, when the office manager copied and pasted employee names from different sources into the spreadsheet, some strings were formatted one like “Scaffidi, Chris” while others were formatted like “CHRIS SCAFFIDI”. In order to put these strings into a consistent format, the user had to manually reformat the data, which was tedious and error-prone.

The rules for validating and reformatting instances of a data category are not specific to a particular user application. Rather, the rules are often implicitly or explicitly agreed upon by society at large, or by an organization’s members. Though an application developer could conceivably include support in an application for automatically validating and reformatting certain extremely common data categories, such as person names, it would be impossible to anticipate or support every single user’s data categories, which include organization-specific job titles, stock ticker symbols, NSF grant numbers, and other domain-specific categories.

Existing approaches for describing data, such as types and formal grammars, are insufficient for automating these users’ validation and reformatting activities. One limitation of these approaches is that they treat values as definitely valid or definitely invalid; that is, they do not support a non-binary scale of validity, as required for these data categories. Because of this and other limitations, existing approaches provide an unsuitable basis for helping end users to automate the validation and reformatting of strings that they encounter during everyday tasks.

In view of the unmet needs of end users and the limitations of existing approaches for validating and reformatting strings, this dissertation presents a new kind of abstraction, called a "tope", which contains functions for validating and reformatting instances of one data category, such as phone numbers. Validation functions are non-binary, in that they can identify strings that are questionable.

This dissertation also presents a supporting development environment, called the Tope Development Environment (TDE), which helps users to implement topes and then use the tope implementations to automate the validation and reformatting of data. Studies show that with the TDE, it is possible to create tope implementations for validating and reformatting a large variety of strings from real spreadsheets and web forms. In addition, experiments confirm that the TDE enables users to validate and reformat data more quickly and effectively than is possible with currently-practiced techniques. Moreover, Tope implementations are reusable across different applications and by different people, offering the potential to save users the time and effort of implementing a tope when they need to validate and reformat data.

1.1 Two difficult activities: validating and reformatting strings

People use software for many different tasks. For example, they use software to fill out expense reports, build employee rosters, coordinate family gatherings, and survey customers. Each of these tasks includes many steps that involve editing, copying, or pasting strings. Each string is a series of characters, where each character is a grapheme such as a digit, forward slash, or period. For instance, at Carnegie Mellon University, filling out an expense report requires going to a certain intranet application and filling out a web form with strings like account numbers, person names, dates, per diem rates, claimed expenses, and descriptions of expenses. From the user's standpoint, even numbers such as per diem rates look like strings on the screen.

Users do more than just edit, copy, and paste strings: their tasks also often call for validating and reformatting strings. For example, a contextual inquiry described later by this dissertation revealed that workers rarely fill out expense reports for themselves. Instead, they typically record their expenses and related information in spreadsheets that they send to administrative assistants, who then laboriously enter that information into the web form on the intranet. As the administrative assistants transfer information from the spreadsheets to the web form, they often encounter strings that look a little odd. For instance, they sometimes see claimed expenses that seem just a little high, and they sometimes see account numbers for grants that should generally not be charged for everyday expenses. Administrative assistants mentally validate the information in order to detect such anomalous values, which they usually double-check by sending an email to the worker who provided the data. Such strings are "questionable": neither obviously valid nor obviously invalid, and therefore very often worthy of double-checking. Moreover, as the administrative assistants transfer information from the spreadsheet to the web form, they often need to reformat strings. Dates are particularly troublesome, since the web form requires dates to be formatted like "Dec-13-2006", and strings provided by workers are rarely in that format.

Unfortunately, end-user applications do not provide any convenient means for automating these validation and reformatting operations, forcing users to perform the operations manually (or mentally, that is, in the case of validation). This is disappointing, since the whole purpose of an application (compared to system software) is to support specific user tasks. Wikipedia describes application software as "any tool that functions and is operated by means of a computer, with the purpose of supporting or improving the software user's work. In other words, it is the subclass of computer software that employs the capabilities of a computer directly and thoroughly to a task that the user wishes to perform." For example, in the case of doing a Carnegie Mellon expense report, the user relies on two different applications (the Microsoft Internet Explorer browser and the Microsoft Excel spreadsheet editor) plus a server-side Oracle application, but none of these applications offers convenient support for detecting questionable account numbers or for putting dates into a format like "Dec-13-2006" (though Excel does support several other date formats). If end users want to reformat strings in their browser, they must retype the strings. If they want to automate these validation and reformatting tasks in Excel, they must write a macro in the unfamiliar VBScript language. This is no minor limitation, since according to one study, 40% of spreadsheet cells contained non-numeric, non-formula textual data [33].

Some applications are intended to help people create programs, and these, too, have insufficient support for creating programs that validate and reformat strings. For example, this dissertation describes a series of interviews with people involved in creating web form programs to collect contact information like person names, phone numbers, and mailing addresses. Some interviewees created these web forms by typing in a text editor, one interviewee used a sophisticated tool called CodeCharge Studio, and one interviewee created a web form by repurposing the "comments" portion of a blogging application.

Regardless which application was used to create them, all these web forms lacked validation on most text input fields (except to check that inputs were non-empty and not bigger than a certain length). When asked why they omitted validation, interviewees explained that they did not want to prevent people from entering unusual but valid strings—that is, questionable inputs. Their reasoning was logical: the only validation technique conveniently supported by web form programming tools is validation of strings based on regular expressions, and regular expressions cannot distinguish among obviously valid strings, invalid strings, and questionable strings (which deserve double-checking before they are accepted). Implementing more sophisticated validation requires writing JavaScript, which would have taken longer than the interviewees wanted to spend. In fact, many interviewees did not even know how to write JavaScript or regular expressions, which is not surprising because approximately half of the interviewees were end-user programmers (people who did not consider programming to be their primary profession). End-user programmers have difficulty understanding and writing regular expressions [16], and not all are capable of the further leap to JavaScript. In the absence of validation, web forms ended up accepting many invalid values, such as "12 Years old" as a street address.

Moreover, interviewees' web forms allowed users to enter each kind of data in a wide range of different formats. For example, users could enter phone numbers in any format desired. As a result, each kind of collected data ended up in a mishmash of different formats. When some programmers wanted to put strings for each kind of data into a consistent format (because they wanted to combine data from multiple sites and remove duplicate values), their programming tools and languages offered no easy way to create programs for reformatting strings.

In summary, applications for end users provide inadequate support for validating and reformatting strings. As a result, end users often perform these operations manually. Applications for creating programs also lack adequate support for creating programs that automatically validate and reformat strings. The resulting data contain invalid strings, as well as valid strings in a mishmash of formats.

1.2 Key traits of data in users' tasks

As demonstrated in the examples above, users commonly need to validate and reformat data with five key traits (Table 2).

Table 2. Five key traits of data that users commonly need to validate and reformat

Key traits
Values are short, human-readable strings.
Valid strings are in a certain data category.
Values can be questionable—neither obviously valid nor obviously invalid.
Instances of data categories can be written in different formats.
Data categories are application-agnostic and often organization-specific.

First, each of the important data values in these tasks is a short human-readable string. Instances of some data categories such as salaries can also be interpreted as numbers, but interpretability as a string is a unifying a trait of the data that users commonly need to validate and reformat. In contrast, for example, music files are not in scope, since instances are not human-readable strings. Users' data values are short enough so that each typically appears in a single GUI widget of a program, such as a spreadsheet cell or a web form text field. In contrast, for example, XML documents such as those conforming to the RSS 2.0 specification are not in scope, since instances are neither short nor human-readable by most end users. As another example, mailing addresses are in the scope of this research in situations where a mailing address is a unitary string (as when appearing in a single widget) but not in situations where a mailing address is broken and scattered across many separate strings (as when appearing in several input widgets). Each data value of interest is a short human-readable string, not a fistful of strings.

Second, each application's problem domain calls for most input widgets each to contain values from a certain category, such as company names or mailing addresses. For example, in the expense report web form mentioned above, one input text field was intended for collecting a person name while another field was intended for collecting an account number. For many such data categories, a value is valid if it refers to what information extraction researchers call a “named entity”—an object in the real world such as a company or a building [26][67]. Other data categories, such as salaries or temperatures, do not refer to physical entities. Valid values in these categories are consistent with physical or social constraints. For example, salaries should be positive and, depending on the job, usually but not always within some range. Regardless of whether categories describe named entities or values meeting physical/social constraints, the categories may overlap. For instance, company names and person names have some values in common (e.g.: “John Deere”), and human ages and MIT building numbers have some values in common (e.g.: “11”). For the purpose of validating a particular string, it only matters whether the string is an instance of the correct data category—it does not matter whether that string also might possibly happen to be a valid instance of some other data category.

Third, many data categories lack a formal specification, so strings might be questionable—neither obviously valid nor obviously invalid. For example, an employer field in a web form might usually refer to a company (a named entity). Yet what constitutes a “real” company and, therefore, a valid input? A company does not need to be listed on a stock exchange or even incorporated in order to be real. Even a private individual can employ another person. This category’s boundaries are ambiguous. In such cases, it is only possible to identify constraints that are usually true, such as the fact that company names usually are title case, usually are no more than a few words long, and rarely contain certain punctuation marks such as dollar signs.

Fourth, values in many data categories can be “equivalent” but written in multiple formats. Specifically, for data categories referencing named entities, two strings with different character sequences can refer to the same entity. For example, “Google”, “GOOG”, and “Google Inc.” refer to the same entity, though they have different character sequences. The first exemplifies the company’s common name, the second refers to the company by its stock symbol, and the third refers to the company by its official corporate title. For categories that do not reference named entities, two strings with different character sequences may be equivalent according to that category’s meaning. For example, a temperature of “212° F” is equivalent to “100° C”. For some kinds of data, there might not be an exact mapping between all formats. For instance, Carnegie Mellon buildings are sometimes referenced with a full name or with an abbreviation (as in “Wean” and “WeH”). But a new building’s name might be chosen months in advance of its abbreviation. Consequently, for some strings like “Gates” (a new building on our campus), there may literally be no equivalent in the abbreviated format. If a user was asked to reformat a string to a format for which no equivalent string existed, it is hard to predict what the user would do in response. He might simply produce the same string (rather than an abbreviation), might produce an empty string, or might invent a new abbreviation. Thus, the reformatting rules used by people for these data categories are not mathematically *complete.*

Finally, many categories are associated with specific organizations, kinds of organizations, kinds of jobs, geographic locations, or other aspects of the problem domain. Examples abound: All staff at Carnegie Mellon University use the same category of accounting codes, and all workers at a manufacturing organization might use the same category of product codes. ISBNs, ISSNs and DOIs are used by employees of most publishing companies (and sometimes by readers of publishers’ web sites and documents). A consequence of the organization- or domain-specificity of data categories is that they are application-agnostic: the rules for the data category come from the organization or the problem domain, not from the particular application used in the solution domain. For example, a string would be a valid Carnegie Mellon account code regardless of whether it appears in a spreadsheet or a web form. Sometimes, organizations have local specializations of more general society-wide data categories. For example, most Carnegie Mellon phone numbers have an area code of “412” and an exchange of “268” or “269”. In this organization, a string like “609-452-1667” could not be a valid phone number.

1.3 Essential requirements for a string validation and reformatting abstraction

Upon contemplation of the five key traits of data that users commonly need to validate and reformat, it becomes clear that the inadequacies of existing applications are deeper than just the feature set exposed to users. The fundamental problem is an inconsistency between the semantics of data in the users’ problem domain and the semantics of data in the applications’ solution domain. Users’ data values *are* strings, but they are *also* something more.

For example, “911-888-9999” is not a valid phone number because that that string could not possibly refer to a real phone number. Valid phone numbers *mean* something, in that they refer to a thing in the real world. These semantics are reflected in a string’s syntactic structure (the clue here being the fact that “911” is not a valid area code). The semantics in the problem domain also explain why it might make sense to reformat “777-888-9999” to “(777) 888-9999”—both of these strings refer to the same phone in the real world.

Applications are blind to these considerations when they just treat data as strings, since the only semantics of a string variable is that it contains a (bounded) series of certain characters. Treating data as just a string fails to capture four of the five data traits identified by Section 1.2: valid strings are in a certain data category, values can be questionable—neither obviously valid nor obviously invalid, instances of data categories can be written in different formats, and data categories are application-agnostic and often organization-specific.

Each of these four traits actually only adds a slight amount of semantics to the simple string abstraction, but together they result in a troublesome mismatch between the string abstraction and the problem domain. This trouble is a specific manifestation of a general problem known to human-computer interaction researchers as a poor “closeness of mapping”. Green explains the concept of “closeness of mapping” by writing, “Ideally, the problem entities in the user’s task domain could be mapped directly onto task-specific program entities, and operations on those problem entities would likewise be mapped directly onto program operations” [37]. In the case of data in users’ tasks, the problem entities have slightly more semantics than are captured by the string abstraction provided by the application.

As a result, there is no direct mapping between the validation and reformatting operations required by users and actual operations provided by strings. Instead, users must map validation and reformatting activities to a series of smaller operations actually provided by the string abstraction (e.g.: operations for reading characters and changing characters). This mapping process thus involves many small manual actions. The bottom line is that this mismatch between abstractions in the problem domain and abstractions in the solution domain actually makes it tedious and error-prone for users to complete tasks.

The closeness of mapping can be improved by providing users with a new kind of software abstraction that is very closely aligned to the problem domain. In the problem at hand, users need to validate and reformat strings, so this new abstraction should provide operations for validating and reformatting strings. That way, when a user wants to validate a string, he can call upon the corresponding abstraction's validation operation. When he wants to reformat a string, he can call upon the corresponding abstraction's reformatting operation.

The key traits identified by Section 1.2 drive several additional requirements for this new kind of abstraction. Since the data values are validated and reformatted based on membership in a certain category, it is reasonable to require that each abstraction should correspond to one data category. Since values in the problem domain might be questionable, each abstraction's validation operation should reflect a non-binary scale of validation in order to distinguish among obviously valid, obviously invalid, and questionable strings.

Since instances of a data category can be written as different character strings, each abstraction must provide reformatting operations as mentioned above. However, one crucial point is the question of what these operations should do when called upon to reformat questionable or invalid strings. The principle of maximizing closeness of mapping indicates that the abstraction should do the same thing that a well-informed user would do: make a best attempt. For example, if an end user were given the putative street address "3043048 Main Street", which is probably questionable due to its really long street number, then the user would have no trouble reformatting it to "3043048 MAIN ST."

In the interest of maximizing closeness of mapping, an abstraction's reformatting operations should do likewise and make a best effort to produce a valid string in the requested format. Note, however, that if the string was invalid or questionable to start with, then the reformatting might not produce a valid string. In fact, because reformatting rules used by people for these data categories are not always mathematically complete (as noted in the previous section), reformatting might not even be able to produce a valid output when given a valid input. Consequently, the user may want to use the abstraction's validation rules to check reformatting rules' outputs.

The last key trait is that users' data categories are sometimes organization-specific and often application-agnostic. Since some categories are organization-specific, application developers will be unable to create abstractions for every user's domain-specific categories. (Application developers also might not even consider it part of their job to provide these abstractions because the data categories are application-agnostic.) Therefore, people inside the organizations must be able to create new abstractions.

Table 3 summarizes these requirements. If an abstraction meets these requirements, then it will suffice for automating the kinds of user activities identified by Section 1.1: when faced with the need to validate or reformat a string, a user can create an abstraction (the fourth requirement) that describes the data category containing that string (the second requirement). The abstraction contains rules for validating and reformatting instances of the data category (the first requirement), so the user can invoke these rules to automatically operate on the data. If the validation rules identify invalid data, then the user knows that an error needs to be fixed. If the validation rules identify questionable data (the third requirement), then the user can choose whether to spend time double-checking whether the error is real. Similarly, the user can invoke reformatting rules and then check the results with the validation rules.

Table 3. Requirements for a new kind of abstraction to help end users more quickly and effectively validate and reformat strings

Requirement	Motivation
Abstraction's expressiveness for *validation & reformatting* operations on *short, human-readable strings*	Presence of these tedious and error-prone activities in user tasks
Abstraction's correspondence to a *data category*	Users' validation and reformatting rules derive from categorical nature of data
Abstraction's *non-binary validation* scale	Presence of questionable values / absence of formal specifications for many categories
Support for *creation by people inside organizations*	Organization- and domain-specificity of many data categories

1.4 Requirements for reusable string validation and reformatting abstractions

It might be helpful if the abstractions were reusable, in the sense that users could call upon an abstraction at multiple points in time (to validate different strings), though reusability is not an essential requirement for the success of this new kind of data abstraction.

Reusability is not an essential requirement because it could conceivably be possible to invent a kind of easy-to-create, "disposable" data abstraction that helps users to very quickly and effectively validate and reformat data. When faced with a certain task involving strings in a certain data category, a user could implement an abstraction, use its operations to validate and reformat the strings, and then discard the abstraction implementation. If the user was not capable of creating the abstraction, he could ask another person from the organization to help create the needed abstraction (which is essentially the strategy that many users successfully employ when they need to create complicated spreadsheets [79]); after applying the abstraction, the user could discard it. In fact, the new kind of abstraction described by this dissertation (Section 1.5 and Chapter 3) is so easy to implement that typical users *can* create new abstractions for themselves, and the abstraction is so easy to apply that users can very quickly and effectively complete tasks, *even without reuse*. Thus, the evaluations described by Section 5.2 and 6.2 show that the primary goal of this research is achieved even without abstraction reusability.

Nonetheless, it would be desirable if each abstraction could sometimes be reused without modification when users encounter instances of the abstraction's data category. That way, while users could always simply create and apply a new abstraction, they also might have the choice to validate and reformat data with an existing abstraction if one is available and if reusing it would be faster or more accurate than creating a new abstraction. Extending this thought, providing users with a way to use existing abstractions as a starting point for creating new abstractions might sometimes make it easier to create new abstractions. In particular, users sometimes like to copy parts of an existing end-user programs and use them as a starting point for creating a new program [20][21][78], when a user is implementing a new abstraction for validating and reformatting strings, it would be ideal if he could copy and combine pieces of existing abstraction implementations.

Based on these considerations and the key traits identified by Section 1.2, several requirements must be fulfilled in order that people can reuse abstractions for validating and reformatting strings. First, since the underlying data categories are application-agnostic, abstractions should not be tied to particular end-user applications. Second, since many data categories are organization-specific or domain-specific, people should have ways of publishing abstraction implementations within or across organizational boundaries (at users' discretion). Third, users should have a way to find a relevant, high-quality abstraction implementation on-demand when they need to validate or reformat instances of some category. Fourth, it should be possible to reuse one abstraction inside another (just as a mailing address contains a city, country, and zip code), as well as combine pieces of multiple existing abstraction implementations while implementing a new abstraction.

Table 4 summarizes these requirements. If the new data abstraction supports these requirements, then one user will be able to publish an abstraction implementation (the second requirement), which another user will be able to find and download (the third requirement), then combine and customize as needed (the fourth requirement) in order to validate and reformat strings in a wide range of user applications (the first requirement).

Table 4. Requirements aimed at enabling people to reuse abstractions for validating and reformatting strings

Requirement	Motivation
Abstraction's *application-agnosticism*	Application-agnosticism of most data categories
Support for *publishing* abstraction implementations (within or across organizational boundaries)	Organization- and domain-specificity of some data categories
Support for *finding relevant, high-quality* abstraction implementation *on-demand*	Inability to anticipate when users will encounter instances of a data category
Support for *combining and editing* parts of existing abstraction implementations when creating a new abstraction	Possibility that some users might have implemented abstractions that do not quite exactly fit the reuse context

1.5 Topes and the Tope Development Environment

Based on the considerations discussed by Sections 1.3 and 1.4, this dissertation presents a new kind of abstraction called a "tope", as well as a supporting suite of many tools. This suite of tools, called the Toped Development Environment (TDE), helps users with implementing, publishing, finding, editing, combining, and reusing topes.

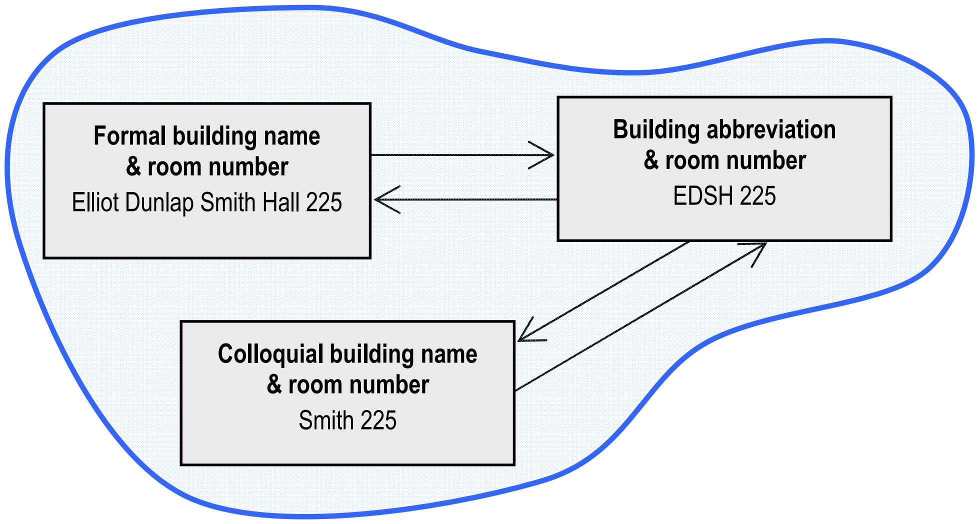
Topes

The word "tope", or Greek for "place", was chosen because each tope describes how to validate and reformat instances of one data category that has a *place* in the problem domain. A tope is a directed graph, where each graph node corresponds to a function that recognizes instances of one format using the same semantics as fuzzy set membership [125], and each edge corresponds to a function that makes a "best effort" attempt to transform strings from one format to another. Figure 1 is a notional depiction of one such graph, which contains three format nodes and four reformatting edges.

Each format is defined by a function `isa:string→[0,1]` indicating whether a certain string is an instance of the format. The semantics are identical to that of fuzzy set theory [125]: though this function's return value is technically not a probability, it is probability-like in that it is an expression of likelihood. As required by Section 1.3, the tope's `isa` functions enable it to identify questionable values. Specifically, the implementation identifies a string `s` as questionable if at least one format does not reject `s` outright, yet `s` does not perfectly match any format, which is to say `0 < max(isa(s)) < 1`.

Each reformatting function `trf:string→string` indicates how a string in one format should be transformed to an equivalent string in another format. Here, "transform" means "to change in form, appearance, or structure" [92], indicating a change in form but not equivalence. A different function would be responsible for the inverse operation.

Figure 1. Notional depiction of a Carnegie Mellon room number tope, including three formats



As explained by Section 1.3, maintaining a very tight closeness of mapping between the abstraction and the problem domain means that reformatting operations must accept questionable or invalid strings, but there is no requirement for them to always produce valid outputs. Accordingly, `trf` functions accept any input string, and they are only required to make a "best effort" to produce an equivalent string in the target format.

Mechanisms outside of the tope model facilitate detecting and recovering in situations where the outputs are not valid (as mentioned shortly).

Each `trf` function is optional: in some topes, some pairs of formats might not be connected by a `trf` function. Because transformations can be chained, a tope's graph of formats does not need to be complete (where each node is directly joined to every other node). The graph-like structure of a tope can cause two forms of ambiguity when reformatting a string: the appropriate route to take through the tope might be ambiguous, and the appropriate starting interpretation (format) of the string might be ambiguous. As a result, the output might not be what the user intended. Another potential problem is that if the graph is not connected, then it might not be possible to find a route through the tope to the target format, resulting in a string that cannot be reformatted. Mechanisms outside of the tope model facilitate detecting and recovering from these situations (as mentioned shortly).

Implementing topes with the Tope Development Environment

Before users can apply a tope's operations to strings, the tope must first be implemented. With the Toped^{++} editing tool in the TDE, users create data descriptions, which are data structures that specify the constrained parts inside of a particular kind of data as well as the different ways in which parts can be concatenated in formats. Based on the user-specified data description for a particular data category, the TDE automatically generates a tope implementation for that data category. For example, the user might create a data description in Toped^{++} for American phone numbers, from which the TDE would generate a tope implementation with operations for validating and reformatting instances of phone numbers.

Internally, the generated `isa` functions test strings with a grammar generated from the data description. Unlike a standard context-free grammar, this grammar contains instructions for identifying strings that are probably instances of the data category, but which violate constraints that are usually true about valid instances of the data category. Specifically, the format's `isa` function returns 0 if the string completely fails to match the grammar, 1 if the string perfectly matches the grammar and all the format's constraints, and a value in between 0 and 1 if the string matches the grammar's basic context-free structure but violates some constraints that are usually but not always true.

In addition, the TDE automatically generates `trf` functions that transform strings from one format to another. Each function permutes and/or transforms parts of the string in one format to generate strings in another format. Each part of the string corresponds to a node in the tree that results from parsing the input string with the grammar of the source format. For example, to reformat the person name "Chris Scaffidi" to "SCAFFIDI, CHRIS", the TDE parses the input string according to the source format's grammar, identifies the parse nodes corresponding to the first and last name, permutes them, capitalizes them, and concatenates them around a comma and a space.

Calling upon topes' operations to validate and reformat strings

The TDE provides more than just a tool for implementing topes. It also includes add-ins for popular end-user applications so that users can invoke tope implementations' operations to validate and reformat strings in those applications. To date, prototype add-ins have been developed for the Microsoft Excel spreadsheet editor, the Microsoft Visual Studio.NET web form editor, and the Robofox and CoScripter web macro tools [52][63][64]. Each add-in passes data values into the TDE (through a programmatic API that wraps tope implementations with convenience functions) for validation and reformatting, then displays results on-screen in an application-appropriate manner. For example, to validate Excel spreadsheets, end users highlight some cells, click a button, and select a tope implementation. The add-in then passes the highlighted cells' strings into the tope implementation's format validation operations, thereby returning an `isa` score for each string. If a cell's string receives a score less than 1, the add-in flags the cell with a little red triangle, so that the user can notice the error and fix it. The add-in offers an option for the user to hide all flags for questionable strings, so that the user can focus on strings that are definitely invalid.

Depending on the end-user application, add-ins provide a button or other UI control so that users can override some or all validation flags. For example, spreadsheet users can remove all validation flags, regardless of whether the strings are definitely invalid (according to the tope) or merely questionable. In contrast, when a tope implementation validates a web form field input, the user who provided the input can generally override validation warnings for questionable inputs but cannot override warnings for definitely invalid inputs (though the programmer who creates the form can configure more strict or lenient settings). Depending on the add-in and the end-user programmer's preference, these overrides can be temporary (applying only to the particular instance of the string that caused the validation warning), or they can be permanently added to a "whitelist" appended to the data description. In this way, users can gradually add exceptional values as they are discovered at runtime.

There are several ways in which using `trf` functions might cause problems. First, because each `trf` function accepts any string as input, it might produce an invalid output. Consequently, any time that a `trf` is called, its output is checked with the `isa` function of the target format. Second, there may be multiple routes through a tope for transforming a string from format to another, so the appropriate route through the tope might be ambiguous. The TDE side-steps this potential problem by generating totally connected topes and always taking a single-step route through the tope. Finally, when reformatting a string, the result might depend on what format is selected as the string's starting format. The TDE automatically detects when ambiguity might affect the result of reformatting, and it generates a warning message so the user can double-check the resulting output.

Reusing tope implementations

Since the add-ins are responsible for all interactions with end-user applications, the actual tope implementations do not need to contain any application-specific code. Consequently, they are application-agnostic and reusable without modification in a variety of applications.

In order to support reuse across multiple users, the TDE includes a repository system where topes can be uploaded and downloaded. Because users are likely to have a few examples of data that they would like to validate or reformat with existing tope implementations, the repository includes a "search-by-match" feature whereby users can look for tope implementations that recognize examples; for example, they could search for topes that match "5000 Forbes Ave.".

The repository includes a variety of "advanced search" features to help users look for tope implementations that are likely to be reusable. These features filter tope implementations based on whether they have certain traits that are empirically associated with highly-reusable end-user programs in general. For example, since many kinds of end-user programs have proven to be more reusable if they have extensive comments written in English, the topes repository offers search features that filter tope implementations based the extent to which they contain extensive comments written in English.

The current repository prototype is "wiki-like", in that all tope implementations are publicly readable and writeable (though of course user- or group-based authentication and authorization could be added in future implementations). In order that users can choose to share tope implementations only within an organization, the repository software is designed to be easily installed on an organization's intranet server. Users can then upload and download tope implementations from organizational repositories or from public global repositories. When creating a new tope implementation, users can combine format specifications from other tope implementations (even if those implementations were downloaded from different repositories). The TDE analyzes tope implementations to detect common inconsistencies that can easily result from combining format specifications, and it presents a summary of any identified problems so that users can fix them.

Table 5 shows the mapping from requirements to features of topes and the TDE.

Table 5. Relationship between requirements and features of topes and the TDE

Requirement	Fulfilled via
Abstraction's expressiveness for *validation & reformatting* operations on *short, human-readable strings*	Tope `isa` and `trf` functions
Abstraction's correspondence to a *data category*	Scope of tope as a package of functions related only to the formats of one data category
Abstraction's *non-binary validation* scale	Return value of tope `isa` functions
Support for *creation by people inside organizations*	Tope development environment
Abstraction's *application-agnosticism*	Separation of concerns: application-specific code is factored out into application-specific add-ins
Support for *publishing* abstraction implementations (within or across organizational boundaries)	Wiki-like repository prototype that can be installed on an organization's intranet server, or on a globally-accessible web server
Support for *finding relevant, high-quality* abstraction implementation *on-demand*	"Search-by-match" algorithm for finding topes that match strings provided by user; model for sorting topes according to how likely they are to be reusable
Support for *combining and editing* parts of existing abstraction implementations when creating a new abstraction	Copy-paste and testing features in TDE

1.6 Evaluations

A series of empirical studies have uncovered many tasks during which users need to validate or reformat short, human-readable strings. To summarize the key points made in the sections above, these strings are valid if they are instances of a certain category, such as university names or corporate project numbers. Many such categories lack a formal specification, with the result that there often are questionable strings that look unusual but yet are valid. These categories are also often multi-format, in that strings of many data categories can be "equivalent" but written in multiple formats. Many of these categories are organization-specific, so developers of applications for end users cannot anticipate and support all of these data categories. Existing applications do not provide support for automatically validating and reformatting these strings. As a result, users have had to do validating and reformatting manually, which is tedious and error-prone. The finished data are sometimes questionable or even invalid, and they are often inconsistently formatted.

In response to users' needs, this chapter has outlined a new kind of abstraction and supporting environment (the TDE) aimed at enabling end users to create abstractions that capture the validation and reformatting rules for commonly-occurring data categories. The TDE includes add-ins that are integrated into user applications, in order to help users apply topes' operations to strings. Separating all application-specific concerns into add-ins is intended to make tope implementations reusable across applications. In addition, the TDE includes a repository aimed at helping users to publish, find, reuse, and combine tope implementations.

As summarized by Table 6, a series of studies has verified that the topes model, as supported by the TDE, meets the requirements identified by Sections 1.3 and 1.4. As shown by the table's rightmost column, these requirements and corresponding evaluations fall into three areas of concern: *expressiveness* requirements related to simply creating useful validation and reformatting operations for a range of commonly-encountered data categories, *usability* requirements related to the bottom-line question of how well implementing and applying topes helps users to validate and reformat data more quickly and effectively, and *reusability* requirements related to helping people repurpose each others' topes in a wide range of applications.

Table 6. Requirements and corresponding evaluations

Requirement	Evaluation	Concern
Abstraction's expressiveness for *validation & reformatting* operations on *short, human-readable strings*	Implemented topes for the 32 most common data categories in corpus of online spreadsheets, with negligible expressiveness problems (Sections 5.2.1, 5.2.2, and 6.2.1)	Expressiveness
Abstraction's correspondence to a *data category*	Correct by construction (Section 3.1)	Expressiveness
Abstraction's *non-binary validation* scale	Evaluated along with the first requirement above, showing that identification of questionable values slightly increased the accuracy of validation, while inclusion of multiple formats led to an even more sizable improvement in accuracy (Section 5.2.2)	Expressiveness
Support for *creation by people inside organizations*	See final requirement, below, describing lab studies where users implemented topes with the TDE (Sections 5.2.3 and 6.2.2)	Usability
Abstraction's *application-agnosticism*	- Reused spreadsheet-based topes on web form data, with negligible loss of accuracy (Section 7.2.1) - Implemented add-ins for several end-user applications, and provided documentation to other research teams who implemented add-ins for web macro tools (Section 7.1.1)	Reusability
Support for *publishing* abstraction implementations (within or across organizational boundaries)	Demonstrated with example (Section 7.1.3)	Reusability
Support for *finding relevant, high-quality* abstraction implementation *on-demand*	- Tested search-by-match algorithm on the 32 spreadsheet-based topes, giving 80% recall accuracy with a query time under 1 second (Section 7.2.2) - Based advanced search filters for identifying reusable tope implementations on an earlier model that could predict with 70-80% recall (at 40% false positive rate) whether a web macro would be reused (Section 8.2.4)	Reusability
Support for *combining and editing* parts of existing abstraction implementations when creating a new abstraction	Explained how the TDE works on examples of combining and editing format specifications from multiple tope implementations to create a new implementation (Section 7.1.3)	Reusability
Enable end-user programmers to more quickly and effectively validate and reformat data	**- Compared user performance for validating 50 strings** with TDE, versus user performance validating data with Lapis system; TDE users were twice as fast and found three times as many invalid strings (Section 5.2.3) **- Compared user performance for reformatting 100 strings** with TDE, versus user performance manually reformatting the data; TDE users were twice as fast, holding accuracy constant (Section 6.2.2)	**Usability**

Expressiveness

By implementing topes and using them to validate and reformat a wide range of spreadsheet and web form data, I have shown that the TDE makes it possible to express useful validation and reformatting operations for several dozen of the most common kinds of string data encountered by users. These evaluations showed that identification of questionable values slightly increased accuracy, while inclusion of multiple formats led to even more sizeable improvement in accuracy.

Usability

Toward verifying that topes and the TDE meet the most important overall "bottom line" requirements of this research, two experiments have confirmed that the TDE enables users to validate and reformat data more quickly and effectively than is possible with currently-practiced techniques. First, a between-subject experiment has examined how well the TDE helps end users to validate strings. With the TDE, users finished their tasks twice as fast and found three times as many invalid strings as with the comparison system, Lapis [70]. Second, a within-subjects experiment has shown that users can implement and use topes to reformat spreadsheet cells, and that this process is so fast that the cost of implementing a tope is "paid off" after only 47 spreadsheet cells (the alternative being to do the work manually, which is the only real option for users right now). Both experiments included a user satisfaction survey, which confirmed that users are extremely satisfied with the topes-based approach and eager to see it deployed in commercial applications for end users.

Reusability

Several evaluations and experiences have confirmed the reusability of tope implementations. First, I reused spreadsheet-based topes on webform data, with negligible loss of accuracy. Second, add-ins have been implemented for several end-user applications, showing that tope implementations can be reused without modification to validate and reformat strings in multiple applications. Third, when tested on tope implementations for the most common few dozen kinds of data, the search-by-match algorithm demonstrated a recall of over 80% and query time of under 1 second, which is sufficient for implementing user interface features that search through the collections of tope implementations that users are likely to have on their computers. Fourth, an analysis of this algorithm shows that it is highly parallelizable and suitable for searching through repositories of many more tope implementations. Finally, a variety of empirical studies show that the tope implementation traits underlying the repository's advanced search filters tend to be useful for identifying reusable end-user programs in general.

1.7 Thesis and implications

Drawing together the empirical results described by the previous section, the following thesis statement summarizes the most important claims supported by these evaluations:

Topes and the TDE are sufficiently expressive for creating useful, accurate rules for validating and reformatting a wide range of data categories commonly encountered by end users. By using the TDE to implement and apply topes, end users can validate and reformat strings more quickly and effectively than they can with currently-practiced techniques. Tope implementations are reusable across applications and by different people.

This research develops an approach for helping users to automate validation and reformatting activities that are now performed manually. This approach is founded on the notion of bringing the abstractions supported by end-user applications into closer alignment with the data involved in the problem domain. Improving this alignment has proven to be an effective way to help users to automate validation and reformatting of strings. An implication is that researchers perhaps could help people with other programming tasks by finding additional situations in which programmatic abstractions have a poor match to the problem domain, then by developing suitable new abstractions with a closer alignment to the problem domain.

As abstractions that are reusable across applications, topes highlight the leverage provided by application-agnosticism. One benefit of such abstractions is that when a user creates an abstraction, he can then reuse it in many applications and possibly share it with other people. A second, even more significant benefit of research aimed at developing application-agnostic abstractions is that the resulting research contributions are highly *generalizable*, as they can be leveraged in a range of end-user programming applications. An implication is that as researchers attempt to develop new abstractions to assist end-user programmers, it would be desirable if those abstractions were application-agnostic rather than confined in applicability to just one programming tool or other application.

One of the keys to topes' success as application-agnostic abstractions is that they are models of the users' "real world" problem domain, which naturally cuts across many different software applications. The implication is that researchers might find it more straightforward to design application-agnostic abstractions if those abstractions are tied to phenomena in the real world (rather than one particular application).

Another key to topes' success as application-agnostic abstractions is the approach's support for incremental modeling of the problem domain. Users can create and use topes on an as-needed basis, without having to make up-front investments in a large ontology or a carefully designed collection of topes. The implication is that incrementalism might be a desirable feature to consider supporting in other application-agnostic abstractions.

A third key to topes' success is the approach's support for semi-automated creation and use of abstractions. Users can specify validation rules that keep out definitely invalid inputs but which allow users to override warning messages for questionable inputs. In addition, transformation rules only need to make a "best effort" and are allowed to produce invalid outputs if appropriate (which the validation rules can detect for double-checking). Thus, users can create and productively use topes even for data categories that cannot always be precisely modeled. The implication is that semi-automation might be a desirable feature to consider supporting in other application-agnostic abstractions.

Finally, perhaps the most significant key to topes' success is that they target a level of semantic granularity appropriate for the problem domain at hand. Specifically, topes capture more semantics than grammars (such as regular expressions), which are purely syntactic, but less semantics than traditional information models (such as the semantic web [9]), which capture extremely detailed relationships among entities. While tope `isa` functions recognize strings based on syntax, they are part of an abstraction intended to model membership in a data category. This shallow semantics is deepened slightly through `trf` functions, which capture the equivalence of strings, usually in the sense of referring to the same real-world named entity. The success of the topes model shows that data abstractions can be practical and effective if they model a shallow level of semantics, thereby retaining usability without sacrificing usefulness for supporting real-world tasks. This result hints that software engineering researchers might be able to find other useful application-agnostic abstractions at a shallow level of semantic detail.

1.8 Outline

Chapter 2 presents a series of empirical studies that have identified a widespread need for better validation and reformatting features in end-user tools. This chapter reviews the key insights provided by these studies, as well as the requirements for helping users to automate validation and reformatting activities. Examining existing approaches that meet fairly large subsets of these requirements shows that no existing approach actually meets all requirements. Based on the identified requirements, Chapter 3 describes a new kind of abstraction, topes. It explores the role and interplay of `isa` and `trf` functions as well as several different ways that topes might be implemented.

Whereas Chapters 2 and 3 lay the motivational and conceptual groundwork for this dissertation, Chapter 4 shifts the focus to tools by giving an overview of the TDE. The subsequent four chapters, Chapter 5 through Chapter 8, dive into the details of four different aspects of the TDE. Chapter 5 focuses on `isa` functions, describing both the TDE's support for implementing these functions and for actually calling them to validate data. Chapter 6 focuses on `trf` functions, including the TDE's support for implementing and calling these functions to reformat data (with an emphasis on handling ambiguity). Chapter 7 describes the TDE's basic support for reuse of tope implementations; it discusses the application-agnosticism of tope implementations, the search-by-match algorithm for recommending tope implementation, and repository features for sharing and reusing tope implementations. Chapter 8 reviews empirical studies which reveal that reusable end-user programs often have certain traits, and it describes advanced repository search filters aimed at finding topes that have traits commonly associated with reusable end-user code

Chapter 9 identifies the key contributions, claims, and conclusions supported by this research and outlines opportunities for future work.

Chapter 2. Motivation

"End users" is the label commonly attached to people who use software in order to complete real-world goals such as communicating with families, running businesses, and so forth [51][78]. But in some cases, achieving these goals quickly and effectively requires end users to actually *create* software to help them with their goals. For example, many teachers create spreadsheets to quickly compute grades, and some marketing specialists create web forms coupled to databases in order to automate the storage of data from potential customers.

In this act of "end-user programming", an end user creates a program or other code as a means toward an end, rather than an end in itself. This contrasts with "professional programming", in which somebody creates software because it is the main deliverable of his job and an end in itself. Because these programs help to *automate* certain activities in user tasks, end-user programs are extremely valuable to the people who create them, despite the fact that these programs are often smaller and less complex than programs created by professional programmers in traditional textual programming languages.

But in entrusting part of a task to a custom program, an end user comes to rely a bit on the program. His life or his work becomes a bit dependent on that program's success. If the program malfunctions, or if creating the program takes far longer than the end user anticipated, then the act of end-user programming can prove to be a waste of time and a source of stress. The same could be said for software in general. By creating hassles for users, software can become a hindrance rather than a help.

Thus, as software becomes increasingly integral to our lives and businesses, there is a rising need to understand and solve the problems that end users encounter when using and creating software. These concerns prompted me to perform a series of empirical investigations aimed at understanding the needs of end users, with a particular focus on users in the workplace.

My series of investigations began with analysis of federal data, which revealed that by 2012, there would be nearly 90 million computer users in American workplaces.

The analysis highlighted another interesting point: the most prominent end-user applications were data-intensive rather than computation-intensive. In particular, workers regularly created and edited databases and spreadsheets, which often contained string data rather than numeric data or computations.

These preliminary conclusions suggested the need for further studies aimed at clarifying the relationship between data-centric applications and users' tasks, in order to identify specific opportunities to help people with their work. I performed a survey, a contextual inquiry, and a series of interviews that together confirmed that data-centric features were indeed widely used.

In addition, these studies showed that users had many difficulties related to validating and reformatting short human-readable strings drawn from data categories such as person names, phone numbers, and university project numbers. These categories often include values that are questionable yet nonetheless valid, and their values typically can appear in multiple formats. The categories are "application-agnostic", in that the rules for validating and reformatting instances are not specific to a particular application (such as Excel). Rather, the rules are implicitly or explicitly agreed upon by society at large or within an organization.

Having understood challenges that end users faced, I considered the available approaches for validating and reformatting this data, but I quickly discovered that they poorly fit the needs of end users, motivating me to develop the new approach described by this dissertation.

2.1 Analysis of federal data: numbers of end users and end-user programmers

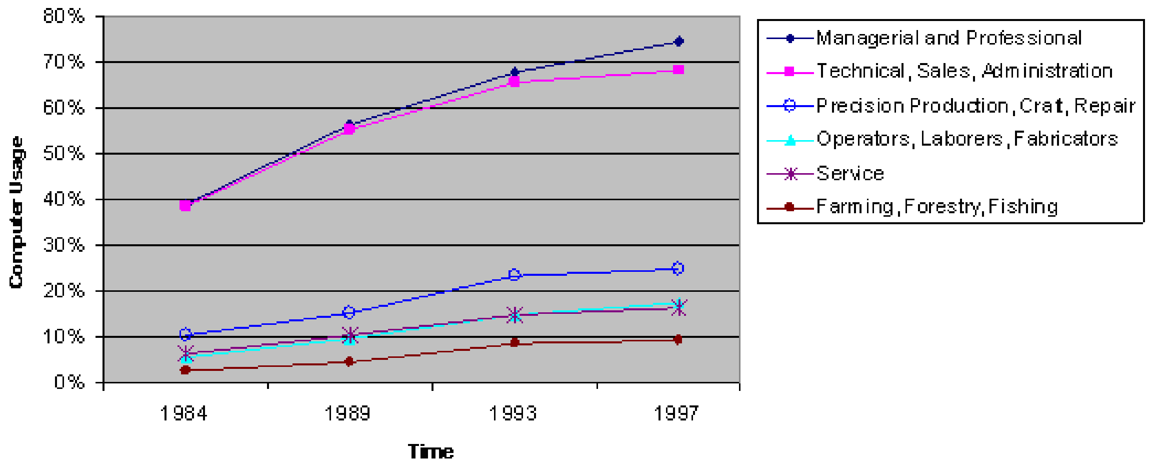
As a first step toward understanding the size and needs of the end-user population, I analyzed government data with an emphasis on answering three key questions about end users: How many are there? What jobs do they have? What kinds of software applications do they use?

How many end users are there?

As first reported in 1995 [17] and widely disseminated as part of the Constructive Cost Model (COCOMO) version 2.0 [18], Boehm et al. predicted that the number of "end user programming performers" in American workplaces would reach 55 million by 2005. This number originally functioned as an estimate of the number of people who would not benefit from COCOMO 2.0, thereby bounding the applicability of that proposed model. Boehm's method depends on a 1989 survey by the US Bureau of Labor Statistics (BLS) that asked American workers about personal computer usage on the job. The method begins by assuming that those usage rates would not change, and it makes the simplification that all end users would eventually perform programming-like tasks. The method then copes with these approximations by incorporating judgment-based multiplicative factors to adjust for the rising usage of computers and the fact that not all end users are programmers. Based on these assumptions, the method yields the prediction of 55 million end-user programmers in American workplaces in 1995.

I developed a more sophisticated model and incorporated more recent BLS data to provide an updated projection of computer use in American workplaces. BLS asked workers about computer usage in the 1984, 1989, 1993, and 1997 Current Population Surveys (CPS) Plotting these data shows rising computer usage trends for each occupational category [42] (Figure 2). As a result, simply using computer usage rates from 1989 under-estimates the total number of computer users.

Figure 2. Computer usage rates rose steeply among American workers since 1984.



The key to extending the prediction method is to model the salient S-shaped curve of each occupation's computer usage, which is particularly apparent for the lower four curves of Figure 2. (The top two curves resemble the right halves of S-curves, with the left halves occurring prior to the first survey in 1984.) This "logistic curve" typifies many diffusion phenomena—ranging from the propagation of a virus through a population to the adoption of technological innovations [97][119]. In each case, a phenomenon of interest initially affects only a small fraction of the population. However, as those people interact with the population, they share the phenomenon (like an infection), causing the incidence to increase until it affects virtually everybody. The rate of increase starts flat (since there are few people promulgating it) and ends flat (since the population is nearly saturated). Although more complex functions exist for certain contexts [119], the simple logistic curve is appropriate in this case where only four data points are available.

A least-squares fit for each occupation's computer usage rates yields a function predicting how usage will develop in the future. As with Boehm's original method, these now-improved usage estimates can be multiplied against projected occupational head counts to estimate the total number of end users in the future in American workplaces. Unfortunately, BLS only issues projections for one year per decade; hence, the only projection available is for the year 2012 [41]. Inserting t=2012 into the fitted functions and multiplying each rate against the corresponding projected head count yields a projection that approximately 90 million people will use a computer in American workplaces in 2012 (Table 7).

This method assumes that computer usage rates can be modeled by a simple innovation diffusion curve. Such an assumption is somewhat suspect, in part because it presumes that the innovation under discussion (here, the computer) does not change substantively during the course of diffusion. This is, of course, not true: computers continually increase in power and utility. Hence, future computer usage rates will likely exceed those indicated by the foregoing model. Therefore, 90 million probably is a lower bound on the number of people who will use a computer in American workplaces in 2012.

Table 7. Nearly 90 million people will use computers at American workplaces in 2012

Occupational Category	Projected Occupational Count in 2012 (in thousands) [41]	Projected Percentage Using Computer at Work in 2012 (fit to[42])	Projected Computer Usage at Work in 2012 (in thousands)
Managerial and Professional	52,030	83.0%	43,209
Technical, Sales, Administration	42,695	72.1	30,804
Precision Prod., Craft, Repair	14,860	29.9	4,442
Service	31,905	19.1	6,098
Operators, Laborers, Fabricators	22,723	21.0	4,782
Farming, Forestry, Fishing	1,107	11.1	123
Totals:	**165,320**		**89,459**

What kinds of work do end users have?

Breaking down this large and growing population according to occupation provides insight as to *what kinds of work* will need to be supported by software. Many of these future users would colloquially be labeled as "information workers"—or, more precisely "knowledge workers", defined by Drucker as someone whose job performance depends on knowledge of subject matter, rather than manual labor [29]. Specifically, as shown in Table 7, nearly 50% of computer users in American workplaces will have managerial/professional jobs, such as those in middle-management, public relations, consulting, human resources, software development, or engineering. Another 35% will have technical/sales/administration jobs, such as those in healthcare, bookkeeping, and basic office functions.

This end-user population vastly outnumbers and will continue to vastly outnumber professional programmers in American workplaces. According to CPS data, the total number of American "computer scientists and systems analysts," "computer programmers," and "computer software engineers" ranged from 2.0 to 2.5 million between 2000 and 2005 [23]. Moreover, BLS projects that the total number of programmers in these categories will remain under 3 million in American workplaces through 2012 [41]. So based on these projections, there will soon be *thirty times* as many computer users in American workplaces compared to professional programmers in American workplaces.

Just as end users in American workplaces vastly outnumber professional programmers in America, end users worldwide also appear to vastly outnumber professional programmers worldwide. This conclusion is based on information provided by the former chief systems architect at BEA, a division of Oracle focused on Java-based development environments [12]. In 2004, he estimated that there were 10 million programmers worldwide using general-purpose textual programming languages, such as Java, but that there were 100 million people worldwide who could “build really complex spreadsheets, do Visio, use Access”. This, of course, does not even include less sophisticated end users. Thus, Bosworth believed that there were *at least ten times* as many end users worldwide as professional programmers.

What kinds of software applications will end users require?

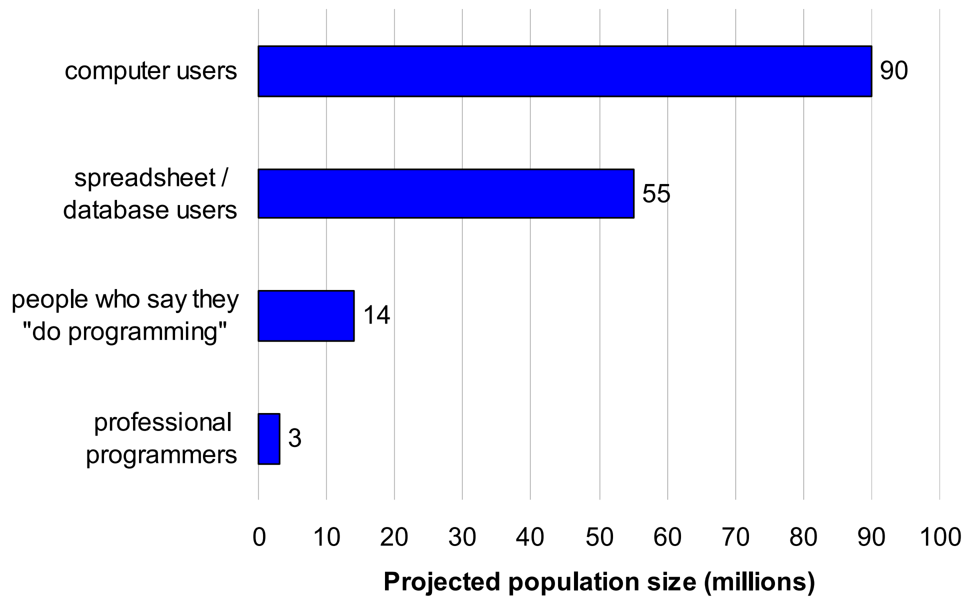
Decomposing the end-user population according to application usage helps to clarify *what kinds of applications* are likely to be required. In particular, this decomposition provides two alternate directions for getting at the relative importance of end-user programming. One direction follows the direct approach of decomposing the end-user population based on what kinds of applications are used, with an emphasis on popular programming tools (such as Microsoft Excel and Microsoft Access). The other direction follows an indirect approach of decomposing the end-user population based on the activities that they do with the applications—in particular, whether users say that they use applications to create programs. It turns out that these two directions yield interestingly different answers.

The direct approach is to decompose the end-user population based on the applications that they use. CPS interviews since 1989 have included a number of questions about application usage. Survey responses show that usage of spreadsheets and databases, in particular, grew steeply throughout the 1990’s. Around 10% of computer users in American workplaces reported “using spreadsheets” in 1989, and by 1997 this had risen to over 30%. Likewise, around 10% reported “using databases” in 1989, and by 1997 this had also grown to over 30%. Over the next four years, usage of these tools continued to explode, with over 60% of American end-user workers reporting that they “used spreadsheets or databases” in 2001. This amounted to over 45 million end users of spreadsheets or databases in American workplaces. If at least the same proportion of people use spreadsheets or databases in 2012, then at least 55 million people will use spreadsheets or databases in American workplaces in 2012.

The indirect approach is to focus on what people say that they do with applications. The proportion of American end-user workers reporting that they “do programming" has remained relatively constant, rising from around 10% in 1989 to only around 15% in 2001 (about 11 million people in American workplaces). If the proportion is at least 15% in 2012, then at least 14 million end users in American workplaces will say that they "do programming".

Thus, while the total number of end users in American workplaces is likely to be around 90 million, the two approaches for estimating end-user programmer counts give projections of approximately 55 million and 14 million people in American workplaces. Figure 3 summarizes the key projections described to this point, highlighting the strikingly small number of professional programmers relative to both estimates of the 2012 end-user programmer population in American workplaces.

Figure 3. Projected population sizes for American workplaces in 2012, based on federal data. Note that the categories overlap.



A prominent feature depicted by Figure 3 is the wide gap between the projected number of spreadsheet / database users and the number of people who say that they "do programming"—the two estimates of the end-user programming population. One likely explanation for this (growing) divergence is that many end users rely on spreadsheets and databases as a place to store information but generally do not create computation-intensive programs in those environments. This explanation is supported by the fact that Hall's study of spreadsheets created by well-educated Australian workers found that only 47% used the "if" function [39], while Fisher and Rothermal's survey of spreadsheets on the web revealed that only 44% contained any formulas at all [33]. These numbers suggest that even using an extremely broad definition of "programming" as the use of conditionals or formulas, only around half of spreadsheet and spreadsheet users are "end-user programmers".

In other words, it appears that the large and growing population of end users relies on software applications for *data-intensive tasks* and less so for automation of computation-intensive tasks.

Yet the government data and prior studies of users' spreadsheets offered few clues as to the precise relationship between users' applications and tasks. Additional studies were needed to give more details to help guide the design of improved application features.

2.2 *Information Week* survey: finding flawed features in applications

In order to help elucidate which features were most heavily used and what struggles end users encountered when using these features, I conducted an online survey asking *Information Week* readers about use of software features during everyday work.

Data collection method and sample characteristics

To recruit respondents, *Information Week* published a link to the survey on their web site and emailed 125,000 randomly selected subscribers who had previously indicated their willingness to respond to surveys. *Information Week* generally runs such surveys every few weeks. Participants were entered into a drawing for one prize of $500 and five of $100 each. Within two months, 831 people completed the survey, which *Information Week* reports is a typical response rate for their surveys.[1]

[1] While this is a very low response rate, which could cause sample bias, it is unlikely to reduce the validity of the primary results of this study. The reason is that these primary results depend on factor analysis, which draws on the inter-relationships of application feature usage rather than absolute proportions, and these inter-relationships are probably relatively insensitive to sample bias. For example, while biasing the sample toward highly skilled users might raise the proportions of all feature usages, it seems less likely to impact the relative difference in usage rates between different features. Empirically, this claim of insensitivity is supported by the validation described below, which shows that there is little internal variability in the analysis results despite the range of different users who did choose to participate in the survey. That is, for the range of different users that we did see, there were only fairly small differences in the relative usage rates among users.

The magazine advertises itself as "exclusively for information managers… involved in computer, communications and corporate planning," so unsurprisingly, 76% of respondents managed at least one subordinate. Only 10% worked for IT vendor firms; instead, respondents generally came from business services, education, government, manufacturing, and finance/insurance. Only 23% were IT or networking staff, while the majority instead were consultants, managers, or administrative staff. Yet respondents generally displayed certain characteristics of programmers. For example, the survey asked respondents about their familiarity with four specific programming terms (variables, subroutines, conditionals, and loops), and 79% were familiar with all four programming terms; in fact, in the past year, 35% actually created all four of the corresponding constructs during their work. Thus, the respondents could generally be characterized as information workers, with a sampling bias that emphasized end-user programmers rather than ordinary end users.

In addition to asking about the demographic data described above, the survey covered 23 application-specific software features related to programming within the context of five particular applications: word processors (including support for macros), spreadsheets, databases, web application design tools, and server-side web scripting environments. For example, the survey asked respondents whether they or their subordinates created database tables, created database views, linked tables via foreign keys, and created stored procedures. Not surprisingly, just as most respondents reported familiarity with the four generic non-application-specific programming terms mentioned above (variables, subroutines, conditionals, and loops), most respondents also reported using many of these application-specific programming features. On average, these application features were used by 50% of respondents.

What software features can be grouped together?

Overall trends in data are easier to discern when the individual data variables can be grouped or clustered. Because the goal of this study was to go beyond analyzing simple application usage (as in the analysis of federal data), it was not enough to simply cluster features based on what applications provided the features. Thus, I clustered software features according to users' inclination to use those features, regardless of which applications provided those features. That is, features were clustered so that people with an inclination to use one feature in each cluster also were inclined to use other features in that cluster, with the recognition that each cluster could contain features drawn from several different applications.

Factor analysis was used to identify these clusters [3][48]. A "factor loading" is an eigenvector of the variables' covariance matrix, such that the vector's cells reflect the correlation of a factor with an observed variable. If a factor has large values in two cells, then the corresponding two data variables highly correlate with the factor (and with each other). For example, an ethnography might record the amount of time that programmers spend in sixty activities, ranging from typing code to talking on the phone, and factor analysis might reveal a "collaborativeness" factor showing that people who often talk on the phone also tend to perform other collaboration activities, such as writing emails and attending meetings.

In preparation for factor analysis, cleaning the data required four steps. First, the 15 respondents with more than 1000 subordinates reported extremely high usage of software features (compared to the other respondents) so I discarded their data, leaving 816 respondents. Second, since factor analysis models relationships among observed data and cannot be applied to unobserved/missing values, it was necessary to filter out all records except the 168 from respondents who reported usage of all five applications. (The other 648 records were retained for validating the factor analysis, below.) Third, I normalized the data according to the "significance" of each software feature. For example, 66% of respondents reported creating hyperlinks, whereas only 16% reported creating static server-side includes. Scaling each feature usage variable to a standardized mean of 0.0 and standard deviation of 1.0 ensured that a "Yes" for a commonly used feature was coded as a value less than 1.0, while a "Yes" for an uncommonly used feature was coded as more than 1.0. Fourth, I normalized feature usage based on the overall usage of each application, since feature usage co-occurs a great deal within each application. For example, people who reported creating JavaScript also tended to create web forms. This "bundling" has nothing to do with abstraction, but rather with the application. Subtracting the average feature usage by each respondent for each application removes this bundling effect, thus revealing inclinations to use features rather than applications.

After cleaning the data, I performed the actual factor analysis to identify the major factors describing the data and to ascertain which variables were strongly related to each factor. I used a commonly-practiced method to identify the major factors and strong variable relationships: running a preliminary factor analysis on all the data, retaining any factors with an eigenvalue exceeding 1.0, retaining only variables with a communality (squared multiple correlation of the variable with all the other variables) exceeding an arbitrary cut-off, and ultimately running a final factor analysis using only the retained factors and variables [48]. As summarized by Table 8, this method yielded 3 factors describing relationships among 19 features, with 4 features removed due to low communality (< 0.1).

Table 8. Software feature use by *Information Week* readers, with the top ten usage rates bolded and factor loadings exceeding an arbitrary cutoff of 0.15 also bolded.

Application	Application Usage (% all users)	Feature Description	Feature Usage (% all users)	Factor Loadings		
				Macro	LinkStruct	Imperative
Slide Editors & Word Processors	96.1	Creating document templates	**73.3**	-0.59	-0.02	-0.17
		Making inter-document hyperlinks	**53.7**	-0.63	-0.04	0.07
		Recording editor macros	33.3	**0.86**	0.03	0.07
		Creating/editing editor macros	30.8	**0.87**	0.07	0.09
Spreadsheets	93.1	Using functions ("sum") linking cells	**89.3**	-0.21	**0.37**	-0.10
		Creating charts in spreadsheets	**80.1**	-0.40	**0.27**	0.14
		Creating inter-spreadsheet references	**66.2**	removed	removed	removed
		Creating spreadsheet templates	50.4	-0.24	-0.19	0.12
		Recording spreadsheet macros	42.3	**0.60**	-0.35	-0.13
		Creating/editing spreadsheet macros	38.6	**0.68**	-0.18	0.02
Databases	79.3	Referencing records by key	**74.0**	-0.02	**0.68**	-0.06
		Creating tables	**71.7**	-0.01	**0.54**	-0.21
		Creating database views	**68.1**	removed	removed	removed
		Creating stored procedures	49.5	0.04	-0.72	**0.22**
Web Pages	68.6	Creating hyperlinks	**65.2**	-0.05	0.06	-0.62
		Creating web forms	**52.2**	0.15	**0.18**	-0.50
		Creating web page behavior scripts	42.9	removed	removed	removed
		Creating JavaScript functions	34.3	-0.04	-0.09	**0.50**
		Using JavaScript "new" function	23.3	-0.03	-0.07	**0.64**
Web Server Script Hosts	52.3	Creating web server script functions	40.9	removed	removed	removed
		Referencing PHP/Perl libraries	27.9	-0.06	-0.41	-0.06
		Using PHP/Perl "new" function	25.4	0.07	0.01	**0.42**
		Using server-side static includes	16.3	-0.09	**0.25**	-0.26

I used two approaches to validate the robustness of the results. First, I repeated the factor analysis using various subsets of the data, such as only including respondents with no subordinates or using alternate factor extraction techniques. In each case, the same qualitative structure appeared. Second, because the factor analysis could only use the 168 respondents with no missing feature usage answers, it was necessary to verify that the factors' qualitative structure generalized to the entire sample by using that structure to construct a traditional scale for each of the factors. For example, the first factor positively correlated with four items and negatively correlated with two items, so its scale equaled the sum of these four items minus the other two. The Cronbach alphas for the scales were 0.82, 0.62, and 0.64, respectively, indicating that the patterns revealed by the factor analysis applied with some consistency throughout the entire data set.

The factor analysis revealed three clusters of features—macro features, data-centric "linked structure" features, and imperative features—such that information workers with an inclination to use a feature in each cluster also were inclined to use other features in that cluster, even though each cluster spanned several applications. The three clusters were the following:

- *Macro features:* The first factor correlated most positively with recording (0.60) and textual editing (0.68) of spreadsheet macros as well as recording (0.86) and textual editing (0.87) of macros in other desktop applications.
- *Linked structure features:* The second factor correlated most positively with creating database tables (0.54), linking database tables via keys (0.68), creating web forms (0.18), creating "static includes" shared by web pages (0.25), creating spreadsheet charts (0.27), and linking spreadsheet cells using simple functions like "sum" (0.37).
- *Imperative features:* The third factor correlated with using Perl and PHP's "new" command in web server scripts (0.42), using JavaScript's "new" command (0.64), creating JavaScript functions (0.50), and creating stored databases procedure functions (0.22).

Template and hyperlink features fell outside all three clusters yet were heavily used, which might suggest the existence of undiscovered yet important clusters.

What group of software features was most heavily used?

Even though the survey had a sampling bias toward end-user programmers rather than ordinary end users, the most heavily used features were data-centric. Specifically, all 10 of the most-commonly used features were related to linked data structures rather than creating macros or imperative programming. These heavily-used features included linking spreadsheet cells with formulas (reported by 89% of respondents), creating charts in spreadsheets (80%), referencing database tables by foreign key (74%), and creating web forms (52%). This contrasted with less-widely-used imperative features, such as creating JavaScript functions (34%) and creating database stored procedures (50%), as well as macro features, including recording word processor macros (33%) and recording spreadsheet macros (42%). These results confirmed the preliminary broad-brush findings generated by the prior analysis of government data.

What problems do end users encounter with software features?

While respondents demonstrated a strong inclination to use data-centric features, they nonetheless reported a number of difficulties with data-centric features in responses to a question asking how the applications have "'gotten in the way' of doing work." Of the 527 people who listed problems in response to this question, 25% mentioned obstacles related to data reuse, especially data incompatibility. (Less common complaints related to reliability, licensing, feature bloat, and general usability problems.) Example comments included the following:

- "Not always easy to move sturctured [sic] data or text between applications."
- "Sometimes hard to 'share' infomraiton [sic] between applications- getting better, but often requires a lot of user-intervention or duplication of data-entry to get work done"
- "Information entered inconsistently into database fields by different people leaves a lot of database cleaning to be done before the information can be used for intended purposes."
- "Compatability [sic] problems betweenformats [sic]"
- "Maintaining output consistency"

These results highlight the need for improving features for “moving” data between applications and reformatting data.

2.3 Carnegie Mellon contextual inquiry: learning what features might help users

The *Information Week* survey provided a great deal of motivation for improving data-centric applications, but it had three limitations. First, respondents’ complaints provided inadequate detail about workflow to identify approaches for achieving improvement. Second, respondents probably were more skilled with programming than end users in general, so their views might not be representative of information workers overall. Finally, end-user programmers may have remembered and reported problems that happened to be particularly annoying but relatively rare, thereby failing to mention problems that are less acute but more chronic (and, therefore, also potentially serious).

In order to address these limitations, I performed a contextual inquiry, a user-centered design method involving the direct observations of users’ tasks [10]. The purpose of contextual inquiry is to gain substantial insight into the limitations of existing applications and to provide guidance for designing better applications.

Data collection method and sample characteristics

I watched three administrative assistants, four office managers, and three webmasters/graphic designers at Carnegie Mellon University for between one and three hours each, in some cases spread over two days. I used a tape recorder and notebook to record information. All participants were recruited by sending emails to university employees listed in the organizational roster as administrative assistants, managers, or webmasters/graphic designers; emails were sent to employees in alphabetical order until ten people agreed to participate. Participants did not receive any monetary compensation.

What key problems did users encounter?

Most subjects used a browser for almost the entire observation period. In particular, the administrative assistants and office managers spent most of their time filling out forms on intranet applications. This task included copying short, human-readable strings such as dates, person names and email addresses between spreadsheets and web forms.

For example, the most common task was that of using an intranet web application to fill out an expense report. Administrative assistants perform this task when a co-worker sends a spreadsheet containing a list of expenses. This spreadsheet typically also contains a Carnegie Mellon University accounting code as well as one or more dates formatted like “12/13/2006” and the name of a locality formatted like “Los Angeles, CA”. The user opens up a web browser, navigates to an intranet application, and fills out a web form. This requires copying the accounting code into the web form, then inputting a date formatted like “Dec-13-2006”, which the user manually must retype because of the formatting difference versus the original format. After filling in the date, each user needed to look up a per diem rate using a government web site, and looking up per diem rates also involved several reformatting operations (such as reformatting the locality).

Manual reformatting is tedious—over the course of the study, I saw participants reformat dozens of dates in this manner. Moreover, manual reformatting is error-prone—I saw participants periodically make typing mistakes and have to retype their entry. There is also the potential (though I did not observe it in practice) that the user might not notice having mis-typed a date during reformatting. There is also the potential (though I did not observe it) that a co-worker’s spreadsheet might specify invalid states or cities in the locality. To detect such errors, the administrative assistant would need to mentally validate the localities and dates before entering them into the web form.

Reformatting and validation also played a role in another example user task, where a manager wanted to create a roster of employees working on a project. To accomplish this task, he visited several intranet applications, each of which listed members of different departments along with their job descriptions, project role, phone number, and other contact information. After copying and pasting data into a spreadsheet, he often had to reformat these short strings, since different sources provided information in different formats. For example, some web sites showed person names in a format like “Chris Scaffidi”, while others showed strings formatted like “Scaffidi, Chris”.

As this user worked, he mentally validated each string (actually moving his mouse over each string as he checked it) and occasionally wondered aloud whether a value was correct. In some cases, he double-checked a value. For example, according to one data source, an employee had an off-campus phone number, which the user confirmed by logging into another system.

What software features might help users?

These observations led to the idea of using a web macro tool such as Robofox [52], Turquoise [16] or C3W [35] to automate these interactions among spreadsheets and web sites displayed in a browser window. Given the intended uses for these tools, it seemed reasonable to think that the tools could automate the process of copying strings from spreadsheets into the expense report form, using those strings to submit the per diem form, and copying the appropriate per diem rate from that web site back into the expense report form. Likewise, it seemed reasonable to think that the tools could automate the manager's task of gathering information from web sites into a tidy spreadsheet.

However, reviewing these tools' capabilities revealed that these tools actually could not automate the study participants' tasks. The reason is that current web macro tools cannot detect if they are copying the wrong string from a web site. For example, the per diem web site is often changed by its webmasters, which could cause a web macro tool to copy some incorrect string (rather than the per diem rate) from the site. But web macro tools cannot differentiate a per diem rate (generally a number between 100 and 300) from any other string. They cannot tell if a string is a valid per diem rate and cannot catch such errors automatically.

Similarly, Microsoft Excel lacks adequate support for automating validation and reformatting rules. For example, the administrative assistant would have to manually verify all fields in the initial spreadsheet, including the Carnegie Mellon University accounting number, date, or locality were valid in the first place. Likewise, in the second task, the manager had to manually check through the spreadsheet for invalid data. While some strings were questionable and deserved double-checking, the user still sometimes decided to use them.

Ideally, end-user applications like web macro tools and spreadsheets would provide a way of automatically reformatting and validating strings. The next few sections will provide additional data points in order to help refine this objective.

2.4 Hurricane Katrina interviews: learning what features might help programmers

The Carnegie Mellon contextual inquiry documented real problems encountered by end users who were less skilled at programming than the *Information Week* respondents. The contextual inquiry helped to address some of the limitations of the *Information Week* survey (specifically, that the survey only provided information about one kind of end user, did not provided detailed information about user problems, and only provided general information about perceived problems and not necessarily about real problems).

Yet all of the problems documented in the contextual inquiry were found specifically in a university environment. The problems mostly involved spreadsheets and web browsers. Moreover, most study participants had zero or minimal programming skill. Consequently, it was unclear whether the problems were unique to universities, to particular applications, or to people with low programming skill (though this last concern was somewhat allayed by the fact that the specific problems found in the contextual inquiry were fairly consistent with the general problems reported by more sophisticated end-user programmers in the *Information Week* survey). It was therefore desirable to perform additional empirical work in order to clarify the generalizability of the contextual inquiry's observations.

The devastation caused by Hurricane Katrina provided an opportune (albeit sad) situation for clarifying the generalizability of the contextual inquiry's observations. In this study, I interviewed people (mostly professional programmers) who had created web applications to collect information about hurricane survivors.

Data collection method and sample characteristics

On Aug. 29, 2005, Hurricane Katrina made landfall near New Orleans, LA. The storm breached levees, cut off phone and electricity service, flooded homes, and displaced hundreds of thousands of people throughout the Gulf Coast. People turned to the web in this tragedy's wake to learn whether friends and family had survived and, if so, where they took shelter. To assist in this search, dozens of teams independently created web sites so users could store and retrieve data related to survivors' locations. Within a few days, one team recognized that the proliferation of sites forced users to perform substantial repetitive work, so this team created "screen scraper" scripts to read and aggregate the other sites' data into a single site. Since these teams seemed to be demonstrating a great deal of altruism by creating these sites for free, it seemed likely that they would be nice people and willing to give me information about the problems that they encountered along the way.

Web site creators were recruited for interviews with a sampling bias toward people who were likely to be end-user programmers. Specifically, of 22 sites located using search engines, 10 were identified that did not display any corporate sponsors' logo, had only a few pages, and lacked extensive graphics. In addition, one aggregator site was identified, to provide insight into the struggles of data aggregation. DNS registration and information on the sites provided the email address and phone number of each site's owner. Six people agreed to participate in one 30-minute semi-structured telephone interview each. Participants did not receive any monetary compensation for participating in interviews.

Three interviewees were managers in information technology vendor firms, one was an office manager at a hot tub retail outlet, one was a graduate student in computer science, and one was a graphic designer at a web developing company. Two people (one IT vendor manager and the graphic designer) actually created a site on their own. Two people (one IT vendor manager and the graduate student) wrote some of the code and then collaborated with professional programmers who finished the coding. Two people (one IT vendor manager and the hot tub office manager) simply wrote requirements and handed them off to professional programmers. Taken together, the interviewees provided a many-faceted view of the web site development process.

To supplement interview data, one person provided the entire source code for his web site. In addition, the client-side code was available online for each interviewee's site.

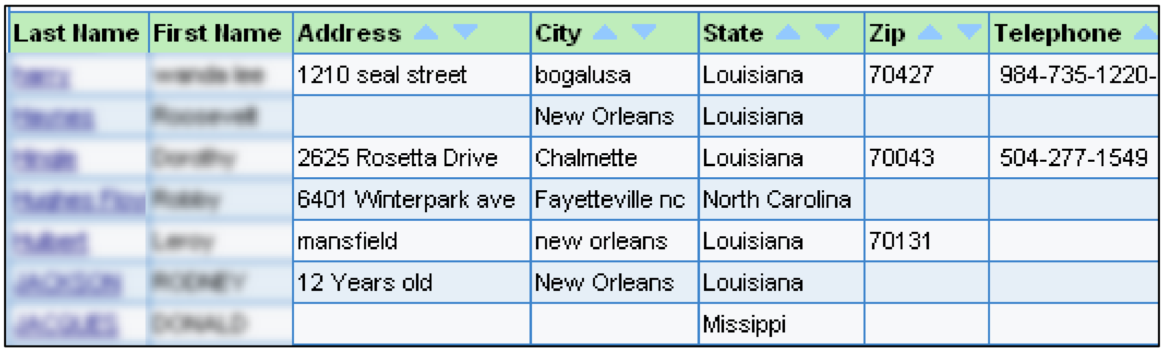
What key problems did programmers encounter?

The implementation of the five web sites created by these six interviewees had little in common. All had one or more web forms for collecting information about hurricane survivors, but these forms were created using different technologies. Three sites were implemented in PHP, one of which was created using a sophisticated web application design tool called CodeCharge Studio. One site was created by repurposing a blog tool: the blog owner posted a blog entry regarding the hurricane and invited readers to use the "comments" feature of the page to upload information about their own status or questions about loved ones who were missing. (Despite its unsophisticated "implementation", the site succeeded in collecting information from 126 users within 2 months of its inception, while garnering 119,713 total views from users.) The fifth and final site was the aggregator site. Like the other four sites, it had a web form so that people could upload information. However, as mentioned above, this Java-based site was primarily populated via scripts that screen-scraped data from other Hurricane Katrina web sites.

Despite the different implementations, these sites' web forms were similar in that they generally did not validate inputs. Some interviewees said that they omitted validation because they thought it so important to collect any data—even possibly erroneous data—that they did not want to put up any barriers to the data input process.

This decision resulted in numerous data errors, some of which were subtle (Figure 4). For example, one web application user put "12 Years old" into an "address" field, which would be hard to distinguish from a valid value such as "12 Years St". As another example, a user entered "C" as a city name. Interviewees stated that they thought it would take too long, or even be impossible, to implement validation that caught such errors yet still allowed unusual-yet-valid inputs.

Figure 4. Invalid data that hurricane survivors entered into a web form that lacked validation



Last Name	First Name	Address	City	State	Zip	Telephone
[redacted]	[redacted]	1210 seal street	bogalusa	Louisiana	70427	984-735-1220-
[redacted]	[redacted]		New Orleans	Louisiana		
[redacted]	[redacted]	2625 Rosetta Drive	Chalmette	Louisiana	70043	504-277-1549
[redacted]	[redacted]	6401 Winterpark ave	Fayetteville nc	North Carolina		
[redacted]	[redacted]	mansfield	new orleans	Louisiana	70131	
[redacted]	[redacted]	12 Years old	New Orleans	Louisiana		
[redacted]	[redacted]			Missippi		

Aggregating data was just as difficult as validating data. Since source sites were built independently from one another, they used differing data formats. In a few cases, aggregators wrote scripts to put data into a common format and to remove duplicates, but in most cases, they meticulously and laboriously performed these operations by hand. Due to this overwhelming amount of effort, they often simply aggregated the data and accepted the fact that it would be a mishmash of formats and duplicates.

What tool features might help these programmers?

Having personally been a professional web application developer for over five years, I found the experience of interviewing these programmers to be surprisingly empathetic and eye-opening. The reason is that as I asked them about their experiences and really listened to the details of what they were saying, I realized that I had personally encountered many of the same problems that they had. However, until I did this interview, I had never been introspective enough about my own experiences to clearly pinpoint the key stumbling-blocks that obstruct the implementation of even basic web sites.

I have created hundreds if not thousands of web forms with a range of technologies, including PHP and Java, but I could probably count on one hand the number of data categories that I regularly validated in my web forms. For example, I generally validated email addresses, dates, and phone numbers with regular expressions or JavaScript. But I never validated person names, mailing addresses, or cities (except for checking that they were non-empty and not longer than a certain length), and I agree with the interviewees that one reason for omitting validation is that existing technologies provide no way for allowing unusual but valid inputs. On the handful of occasions when I wrote screen scraping software, I also had to write a great deal of code for reformatting strings (or left it for the customer to do manually in situations when the budget provided inadequate man-hours for writing such code). It appears to me, based on my memories, that my fellow programmers over the years had essentially the same practices as mine in terms of writing validation and aggregation code.

Programmers like my interviewees (including me, in my previous career) could have benefited from having an easy way to validate and reformat a wide range of different kinds of strings.

Ideally, in order to reduce data errors without erecting insurmountable barriers to data collection, it would have been helpful if the web application had taken a graduated response to "questionable" inputs. For example, the application could have checked whether street addresses ended with the more common street types (such as "Str", "Ave", "Dr"), noticed unusual street types and given the web application user a warning such as, "This does not look like a valid street address. Are you sure that you want to enter '12 Years old' as a street address?" This would allow users to override the system with inputs that have unusual street types (such as "12 Years Upas", whose street type of

"Upas" seems uncommon but happens to be the official abbreviation specified by the US Postal Service for "Underpass"). Examples like these helped to motivate the requirement for distinguishing between questionable and definitely invalid inputs (Section 1.3).

In addition, aggregators could have benefited from having a way to automate reformatting data into a consistent format, in order to identify duplicates. Automatic reformatting could also have prevented each site (individually) from having a mishmash of inconsistent formats. These considerations helped to motivate the corresponding requirement discussed by Section 1.3.

Whereas the Carnegie Mellon contextual inquiry described by Section 2.3 involved some organization-specific categories of data, none of the data involved in the Hurricane Katrina interviews were organization-specific. Instead, data like phone numbers, street addresses, and city names are all widely used and understood across American society. Thus, there is little conceptual reason why every single web developer should have to create an abstraction for these data categories. Ideally, it would be desirable if individuals could share, find, and reuse validation and reformatting code that have already been created by other people, as discussed by Section 1.4. When programmers create multiple forms for collecting data on intranet sites (as I did), there might also be a benefit to reusing some validation and reformatting rules at an intra-organizational level.

2.5 Interviews asking information workers to describe data

Previous projects aimed at supporting end-user programming have benefited from first studying how people describe and reason about programming. For example, the insights developed during these studies have inspired specific language features for helping young children to write programs as well as tool features for helping older students to debug programs [77].

Inspired by the prior success of this method in end-user programming research projects, I applied it to the problem of understanding how users describe common kinds of strings that might need to be validated or reformatted.

Data collection method and sample characteristics

To learn how end-user developers describe data, four administrative assistants were asked to verbally describe two types of data (American mailing addresses and university project numbers) so that a hypothetical foreign undergraduate could find those data on a computer. This syntax-neutral phrasing was intended to avoid biasing participants toward or away from a particular notation (such as regular expressions).

Information workers were recruited by emails sent to administrative assistants in the Carnegie Mellon University employee roster. They were contacted in alphabetical order until four people agreed to participate. One of these four had previously participated in the Carnegie Mellon contextual inquiry. No participants received any monetary compensation.

How do information workers describe data?

In every case, 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 consistently 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, again highlighting the possibility that data might be questionable but not definitely invalid.

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

These results have an important implication. End users appear to be capable of verbally describing these two kinds of data, even something as complicated as a mailing address. It is noteworthy that they even identified and described constraints that were sometimes but not always true. Their answers show that they were keenly aware of the existence of unusual values that were questionable yet still valid.

If these administrative assistants could give these descriptions, including rules for identifying questionable values, then it also seems likely that the web developers in the Hurricane Katrina interviews also could give similar descriptions, including rules for identifying questionable values. Therefore, when the web developers said that they omitted validation because it was too hard to implement validation that would allow unusual inputs, these difficulties were probably not caused by an inability to express appropriate validation rules (in English). Rather, the problem was more likely caused by an inability to implement the validation rules that they knew. Reflecting on the many programming technologies that I used while a professional programmer, I am quite certain that I have never seen a web application development tool that allowed me to implement validation rules that distinguished among valid, invalid, and questionable inputs.

That is, the problem was not with the people but rather with the programming tools' lack of support for concisely expressing validation and reformatting rules. Moreover, the structure of users' data descriptions—as a hierarchy of constrained parts—provides a starting point for designing appropriate tools for helping end users.

2.6 Requirements for features to help users automate validation and reformatting

The studies above have provided the following insights:

- Many computer users could be labeled as information workers (Section 2.1).
- Among information workers who are relatively skilled end-user programmers, the most commonly-used applications are data-centric, and when application features are clustered in a cross-application manner based on users' inclination to use those features, the most commonly-used application features also turn out to be data-centric (Section 2.2).
- The job tasks of information workers often require validation and reformatting operations, which are presently done manually (Section 2.3).
- The categories of data involved in these tasks are often short, human-readable strings. Many of these strings can be written in multiple formats, and some can

appear questionable yet still turn out to be valid. A large fraction of these are organization-specific kinds of data (Section 2.3).

- These same traits are typical of data collected by web forms created by professional programmers. Yet these programmers lack efficient ways to implement code to distinguish between invalid, valid, and questionable inputs, as well as to reformat strings (Section 2.4).
- Some categories of data appear to be "application-independent". That is, certain kinds of data such as phone numbers appeared in the Hurricane Katrina study's web forms, yet they also happened to appear in spreadsheets during the Carnegie Mellon contextual inquiry (Sections 2.3 and 2.4).
- When prompted to describe these kinds of data, end users describe them in terms of a series of constrained parts. They seem to have no difficulty explaining constraints that are usually satisfied but might be violated by questionable-yet-valid data (Section 2.5).

Together, these results point toward an opportunity: to *capture rules* that users already know for validating and reformatting strings, and then to *invoke those rules* as needed to automatically validate and reformat strings.

Capturing the rules requires a way of expressing them as computer-executable instructions. Theoretically, there are many possible approaches for doing this, and the next section will examine the leading contenders. But regardless of the approach, it will be necessary to meet the following requirements (which have already been discussed in some detail by Section 1.3).

Creatable (and usable) by workers "on-site" at organizations: Many strings are from organization-specific data categories, and it would be unreasonable for applications like Microsoft Excel to ship with validation and reformatting code for every single organization's kinds of data. Therefore, somebody "on-site" at each organization would need to express rules for validating and reformatting organization-specific kinds of data. Ideally, this would be the end users themselves, since they come upon the data in the course of daily work and probably would not want to wait for some other person in the organization to implement validation and reformatting code on their behalf. In order to achieve this goal, it might be desirable if the notation for expressing rules also matched the constrained-parts mode of description demonstrated by users.

Expressiveness for "non-binary" validation: Programmers need a convenient way to create code that distinguishes among obviously invalid strings, questionable strings that deserve double-checking but that still could be accepted, and obviously valid strings that should be accepted without hassling the user. While collecting data into spreadsheets, end users need a convenient way to identify strings that are questionable or invalid and should be double-checked and perhaps replaced with different values. Validation will fall short of these needs if it is "binary" and can only categorize data as valid or invalid.

Expressiveness for reformatting rules: Programmers and end users need ways of creating rules that put many strings of the same kind into a consistent format. In particular, programmers need a way of putting data into a consistent format so that they can then identify duplicate values. Thus, programmers and end users need a way to create rules that automatically transform data between different formats.

Reusability: The rules for validating and reformatting data are typically application-independent and implicitly or explicitly agreed upon by society at large or within an organization. As explained by Section 1.4, reusability is, strictly speaking, not an essential requirement; however, it would be desirable if people could reuse validation and reformatting rules in different applications and could reuse rules created by other people in the same society or organization. Moreover, organizations could publish implementations of abstractions for data categories that the organizations "own" (such as mailing addresses as defined by the United States Postal Service, or the URLs as defined by the W3C).

2.7 Related work

No existing approach meets all of the requirements identified by Section 2.6, though some approaches meet subsets of the requirements. The subsections below describe the existing approaches that meet fairly large subsets of the requirements.

2.7.1 Type systems

"A type system is a tractable syntactic method for proving the absence of certain behaviors by classifying phrases according to the kinds of values they compute" [87]. The positive claim implicit in this definition is that classifying phrases according to their kinds of values makes it possible to prove the absence of certain undesirable behaviors.

This definition of type systems is actually broader than the one familiar to most programmers: When a programmer speaks of a "type system", he probably is referring to a nominal type system, which is a type system where the programmer explicitly associates named types with variables. More generally, type systems can be based on other properties of variables, expressions, and other phrases in code; for example, type systems can be based on the structure of records or the set of operations exposed by an object [87]. In each kind of type system, a compiler propagates types through expressions and other syntactic constructs in order to identify code that attempts to call invalid operations.

While professional programmers are perhaps most familiar with type systems enforced at compile-time, researchers have developed several type systems enforced at runtime. For example, dependent type systems allows a programmer to express rules that validate values before they are assigned to variables [83]. From these rules, the compiler generates instructions that check the values, and if they fail to pass the rules, then the instructions throw an exception. Supporting runtime type checking need not entirely sacrifice compile-time type checking, since the compiler can also statically check many rules at compile-time to see if they might ever be violated at runtime. For example, if `x` and `y` are declared to be positive integers and `z` is declared to be a negative integer, then the compiler would detect that `z = x + y` is an invalid assignment; the compiler would also generate instructions that would throw an exception if, at runtime, `z` was assigned from user input, and the user entered 3. The range of constraints supported by these systems is fairly limited. In particular, they seem to provide only very minimalist support for validating strings (represented as arrays of characters). In addition, run-time type systems share two limitations with other type systems.

First, type systems make binary distinctions—either an expression is or is not an instance of a type. This is the source of a type system's strength, as it enables a compiler to definitively determine whether code attempts to perform certain illegal operations.

However, for the purposes of helping end users with the tasks in the studies described above, the binary nature of type checking is a weakness, as it leaves no room for questionable values that nonetheless are valid and should be allowed.

Second, while type systems can categorize expressions according to their types, their only support for reformatting-like operations is through implicit type conversion and type-casting. For example, most type systems allow code to assign integer values to floating point variables (with the actual conversion of the value handled at runtime by a mechanism outside the type system), and the reverse is allowed through an explicit type-cast. Type systems do not provide appropriate string-reformatting operations involved in the problem at hand, such as converting between the names and abbreviations of American states (e.g.: “Florida” to “FL”). Instead, it is necessary to go outside the type system and use a formatting library, such as the library provided by Java for formatting numbers and dates. Such libraries still provide no support for reformatting many kinds of data that end users encounter (e.g.: converting between the names and abbreviations of American states).

2.7.2 Dimension-unit analysis

General dimension-unit analysis systems have been incorporated into at least two professional programming languages (such as Java [2] and ML [46]). These analysis systems could, for example, detect if code attempted to add a variable containing a distance to a variable containing a weight.

Researchers have incorporated dimension-unit analysis into spreadsheet editors. For example, the σ-calculus associates types with spreadsheet cells (based on the placement of labels at the top of columns and at the left end of rows), and the calculus specifies how these types propagate through the spreadsheet. If two cells with different types are combined, then their type is generalized if an applicable type is available (e.g.: “3 apples + 3 oranges = 6 fruit”), or else an error message is shown [31]. Slate performs similar data validation using physical units and dimensions, such as kilograms and meters, in order to detect spreadsheet errors [27]. As in the constraint-inference systems mentioned above (Section 2.7.4), these dimension-analysis systems are semi-automatic: they identify likely errors, but the user of the spreadsheet editor has the option to disregard warning messages.

These approaches are applicable to a wide variety of numeric data, though not to string-like categories of data mentioned by Section 1.2, nor are the systems application-independent (being tied to spreadsheet environments, Java, or ML).

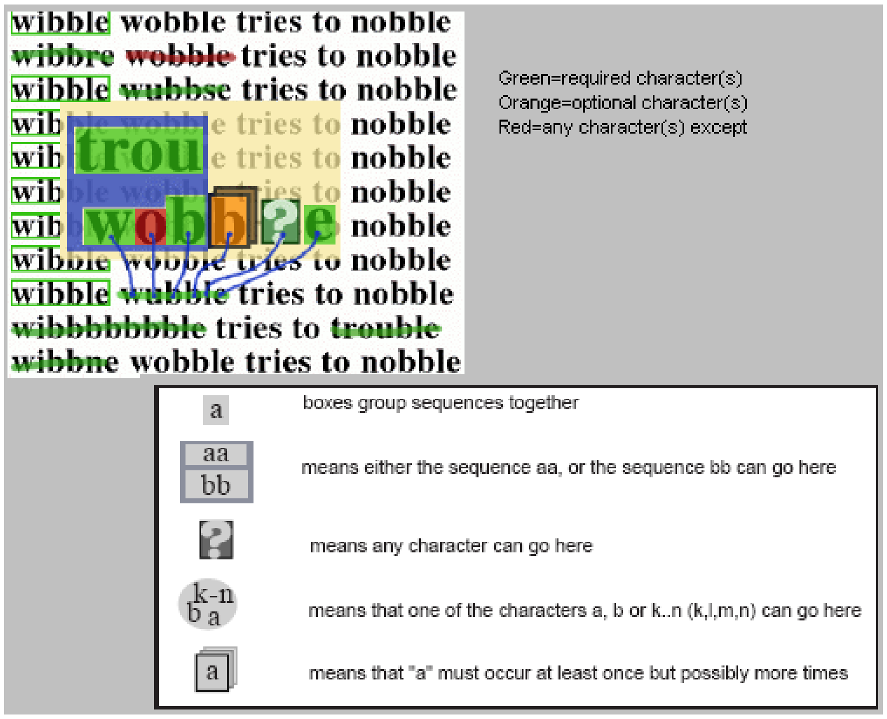
2.7.3 Grammars and other formal languages

Several end-user programming tools allow people to use regular expressions and context-free languages to validate data. These tools either provide a user-friendly visual language overlaid on the underlying grammar notation (thereby sparing users the need to understand the notation), or they add "syntactic sugar" making it easier for users to work directly with grammars.

SWYN is a visual language that enables end-user programmers to manipulate symbolic pictures that are represented internally as regular expressions (The example provided shown below, corresponds to `(trou|w[a-n,p-z]b)b*[a-z]e` (It appears that no screenshots of SWYN expressions for common data such as dates or email addresses have been published.) Empirical studies show that SWYN helps users to interpret regular expressions more effectively than they can with the usual textual regular expression notation.

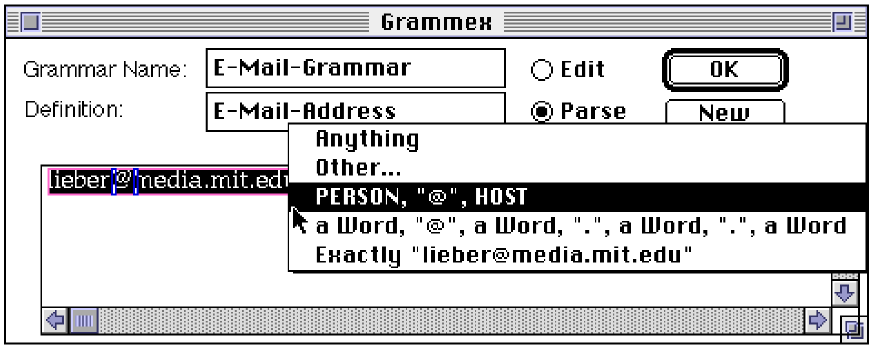
Figure 5) [16]. The example provided shown below, corresponds to `(trou|w[a-n,p-z]b)b*[a-z]e` (It appears that no screenshots of SWYN expressions for common data such as dates or email addresses have been published.) Empirical studies show that SWYN helps users to interpret regular expressions more effectively than they can with the usual textual regular expression notation.

Figure 5. SWYN user interface for defining regular expressions, with notation key


wibble wobble tries to nobble
wibbre wobble tries to nobble
wibble wubbse tries to nobble
wibble wobble tries to nobble
wibble wabble tries to nobble
wibbbbbbble tries to trouble
wibbne wobble tries to nobble
Green=required character(s)
Orange=optional character(s)
Red=any character(s) except
boxes group sequences together
means either the sequence aa, or the sequence bb can go here
means any character can go here
means that one of the characters a, b or k..n (k,l,m,n) can go here
means that "a" must occur at least once but possibly more times

Grammex is an editor for context-free grammars (Figure 6) [62]. The end-user programmer gives examples, shows how to break the examples into grammar non-terminals, and then specifies the form of each non-terminal (for instance, as an integer or a word, or as a list of terminals or non-terminals). In the screenshot, HOST and PERSON were already defined (in dialog windows), and the end-user programmer is specifying an email address grammar. The usability of Grammex has not been evaluated empirically.

Figure 6. Grammex user interface for defining context-free grammars



Apple data detectors are also context-free grammars, though in this case, end-user programmers define a grammar by typing the productions for non-terminals, which (for convenience) can take the form of a regular expression (Figure 7) [79]. The example date format shown below was actually more complicated as published in [79] (to allow for dates with textual month names and other format variations), but has been simplified here to conserve space. The usability Apple data detectors apparently has not been evaluated in laboratory experiments, but the technology has been successfully commercialized in several versions of the Apple operating system (including Leopard).

Figure 7. Apple data detector grammar

```
Date =
{
        (MonthNumber [/\-]+ Day [/\-]* Year?)
}
MonthNumber =
{
        [1-9],
        ([1] [012])
}
Day =
{
        [1-9],
        ([12] [0-9]),
        ([3] [01])
}
Year =
{
        ([1-9] [0-9]),
        ([1-9] [0-9] [0-9] [0-9])
}
```

Finally, Lapis data patterns are similar to context-free grammars, again typed by end-user programmers, though primitives are also provided for common forms such as numbers, words, and punctuation (Figure 8) [70]. The example pattern shown below was actually more complicated as published in [70] (to allow for dates with textual month names and other format variations), but has been simplified here to conserve space. Laboratory user studies have shown that Lapis patterns offer an effective, usable mechanism for selecting valid strings and identifying invalid strings. In particular, Lapis patterns are integrated with a powerful outlier finding method approach for finding invalid strings (discussed further by Section 2.7.4) as well as an efficient method for editing multiple valid strings simultaneously (discussed further by Section 2.7.6).

Figure 8. Lapis pattern grammar

```
@DayOfMonth is Number equal to /[12][0-9]|3[01]|0?[1-9]/
          ignoring nothing

@ShortMonth is Number equal to /1[012]|0?[1-9]/
          ignoring nothing

@ShortYear is Number equal to /\d\d/
          ignoring nothing

Date is flatten @ShortMonth
          then @DayOfMonth
          then @ShortYear
          ignoring either Spaces
                    or Punctuation
```

Though specialized for strings and aimed at end users, these systems share the same limitations as type systems. Data formats in these systems specify binary recognizers: either a value matches the format, or it does not. Moreover, these systems only describe how to recognize strings, not reformat them. Finally, it is cumbersome (though not impossible) in these grammars to express many common constraints on data. For example, it is extremely difficult to express numeric ranges (e.g.: a date's day should be in the range 1 through 31) and negation over runs of characters (e.g.: phone numbers' area codes never end with "11").

2.7.4 Outlier finding and other approaches based on inferring constraints

Outlier finding refers to a general class of techniques aimed at automatically or semi-automatically identifying data points that are inconsistent with the rest of the points in a dataset [6]. In essence, these techniques allow a user to provide examples of data to validate, the system infers a constraint-like description of the data, and then the system identifies data points that do not satisfy the description.

For the most part, outlier finding has been used to validate numerical data. For example, Cues infers constraints over numeric data from web services [93], and Forms/3 is a spreadsheet editor that can infer constraints over values in cells [25]. The systems then check numbers to see if they match the inferred constraints. These systems provide user interfaces and testing methods so that end-user programmers can review, debug, and customize the inferred constraints. From a conceptual standpoint, it is necessary to generalize these algorithms for validating numeric data to also include the string-like data characterized by Section 1.2.

A handful of outlier-finding systems operate on string-like data. For example, the Lapis system mentioned earlier infers patterns, then uses them to find unusual strings that do not match the format very well [70]. Other systems infer context-free grammars [45] and regular expressions, then use them to check strings [111]. Although the inclusion of an inference algorithm greatly simplifies the process of validating inputs, the underlying grammar still suffers from the limitations described above, such as the lack of support for identifying questionable values.

2.7.5 Abstractions for sets of non-string entities in end users' programs

The aforementioned systems based on types, grammars, and constraints essentially embody approaches for identifying *sets* of entities. Types identify sets of expression values that support certain operations, grammars identify sets of strings with certain structure, and constraints identify sets of values that meet syntactic or semantic requirements. By referring to sets of entities rather than individual set elements, types, grammars, and constraints *abstract* away the specific identities of the set elements. Thus, for example, a line of program source code can access a variable's operations based on the variable's type (irrespective of what value actually appears in the variable at runtime), a web form can include a text field that should contain a string that matches a certain regular expression (with the specific value undetermined until runtime), and Forms/3 can propagate measures of correctness through the spreadsheet based on how well various cells satisfy their constraints (meaning that if cell B references cell A in a formula, and cell A fails to satisfy its constraint, then the correctness of cell B is impugned regardless of what specific value appears in A—it is only the degree to which the constraint is violated, rather than the specific value, that is needed to propagate correctness).

Other programming environments also include abstractions for referring to sets of entities. For example, in the SmallStar, Pursuit, and Hands programming environments, user code can programmatically identify objects whose properties satisfy user-specified constraints [40][73][84]. In Gamut and Hands, the user interface displays piles of cards as visual graphical containers of objects [68][84]. (While these containers can be populated programmatically in Hands, they must be statically populated at design-time in Gamut, as the programmer drags and drops icons into the container.)

Because these systems provide for abstractions identifying sets of entities, they bear some similarity to types, grammars, and the constraint systems describe in the previous sub-section. Specifically, in SmallStar, Hands, and Gamut, these abstractions identify sets containing graphical objects, while in Pursuit, the abstractions identify sets containing files. That is, none of them contain abstractions for sets of strings, and therefore they do not solve the problem of helping users to validate and reformat strings.

2.7.6 Simultaneous editing and editing-by-example

Potluck [44] and Lapis [70] support simultaneous editing, in which a user types a change on one string, and the system automatically performs similar edits on other strings. However, these edits are not stored as rules that can be later executed on other data. Consequently, there is no opportunity for reuse. Nix's editing-by-example technique is similar to simultaneous editing, and it infers a macro-like program that can be replayed later. However, this technique does not support validation [80]. In short, none of these systems provides integrated support for reusable validation and reformatting rules.

2.7.7 Federated database systems and data cleaning tools

A federated database is an assembly of relational databases that are generally created, owned, and managed by autonomous groups of people [115]. Because of this level of autonomy, integrating data from the constituent databases often requires validating and transforming their data. These transformations sometimes go well beyond the simple reformatting operations that end users encountered in the studies described above. For example, values might undergo schema transformations as well as filtering and complex arithmetic operations (including computation of aggregate sums or other statistics) prior to combination with data from other databases. Researchers have provided a wide variety of tools and other software infrastructure to help skilled database administrators create scripts and other programs that perform filtering and other computations; these programs typically execute online in order to validate and transform data in response to user queries and similar relational database operations.

Researchers have also provided tools to simplify offline data cleaning and data integration when combining existing data sources into a single database. Unlike in a federated database, the resulting database is then decoupled from the initial data sources. One example tool is the Potter's Wheel, which presents database tables in a spreadsheet-like user interface, allowing administrators to edit specific cells manually to correct errors or to put data into a consistent format [91]. From these edits, the Potter's Wheel infers transformation functions, which administrators can save, reload, and replay on other data values that require the same cleaning steps.

Federated database systems and data cleaning tools are specifically designed for use on relational data, limiting their usefulness for validating and reformatting data in web forms, spreadsheets, and other end-user tools. In addition, the user interfaces of the systems mentioned above are extremely complex, so while they are suitable for use by skilled professional database administrators on complicated database integration tasks, they are probably less appropriate for use by ordinary end users such as administrative assistants.

2.7.8 Value objects and domain values

In object-oriented system design, a "value object" is an object that has no identity [30]. This contrasts with entities, where identity matters. To elucidate the difference, Evans gives the example of mailing addresses [30]. In an e-commerce system's design, it does not matter if an order today is going to the same address as an order did yesterday; the mailing address serves is a value object, since its identity does not matter. In contrast, in a system for managing electrical company service, two customers should not be able to simultaneously register for service at the same address; in this case, the mailing address is an entity, since its identity does matter (as two addresses might need to be detected as equivalent).

The notion of a value object is important in system design because variables containing value objects are almost always attributes of entities. Thus, they generally are implemented as members of objects or columns in databases. The notion of a value object is relevant to this dissertation because, in terms of surface syntax, value objects generally refer to instances of the same categories that users commonly need to validate and reformat. Examples include phone numbers, dates, person names, and so forth.

However, despite this surface similarity, value objects are exactly the *opposite* of what is needed to help users automate reformatting operations. The reason is that reformatting depends on transforming a string from one format into an equivalent string in another format. Usually, this equivalence is a direct consequence of identity (as when two mailing addresses refer to the same building). This is precisely the semantics that value objects *give up*, compared to entities. Thus, users' validation and reformatting tasks bear much more resemblance to named entities (described by Section 2.7.9) than they do to value objects.

A related concept is that of the "domain values pattern", which is a design pattern for implementing domain-specific string-validation rules in object-oriented systems [126]. For example, suppose that a system involved interest rates. The domain values pattern prescribes that the programmer should implement an `InterestRate` class that provides an `IsValid(float)` method and an `IsValid(string)` method. The first of these "static" class methods would return a Boolean indicating whether the specified floating point number is a valid interest rate (e.g.: to check if the value is positive); the second validation method would return a Boolean indicating if the string could be parsed into a floating point number that is a valid interest rate. In addition, the pattern prescribes that the `InterestRate` class should include `InterestRate(float)` and `InterestRate(string)` constructors; each of these would call the corresponding `IsValid` method during object construction and throw an exception if presented with an invalid input. Finally, the domain values pattern prescribes that the `InterestRate` class should provide a `ToString()` method for converting an `InterestRate` instance into a string, as well as (optionally) "static" class methods for iterating over all possible values (though of course the iterator would be omitted when there are an infinite number of values).

In short, the domain values pattern prescribes the creation of a specific class that can be instantiated from strings, can validate these strings, and can serialize instances to strings. Not all domain values are strings, but the pattern assumes that the values can be represented as strings. The level of semantic granularity is very similar to that of the data categories involved in users' tasks.

However, the domain values pattern is limited in the support that it could actually provide for users' tasks. First, it only allows binary validation, so the prescribed classes would be unable to differentiate among obviously invalid values, questionable values, and obviously valid values. Second, the `ToString()` method only supports one format, which would not adequately capture the range of formats in a multi-format data category. Finally, as the pattern calls for implementation of a class, it is somewhat tied to object-oriented programming environments (though the general concepts described in the pattern certainly would generalize to non-object-oriented programming environments).

In addition to the limitations above, the value object and domain value concepts have only been explored to date in the context of system design by professional programmers. But Section 2.6 requires that *end users* should be able to create necessary abstractions for themselves. Therefore, while value objects and domain values offer useful insights as to how professional programmers solve similar problems, it will be necessary to generalize these concepts in a way that supports the creation and use of abstractions by end users across a range of applications.

2.7.9 Information extraction algorithms

Information extraction systems examine natural language text and identify instances of certain data categories commonly referred to as "open lexical items" or "named entities" [4][26][67]. One of the largest and most successful systems for identifying named entities in text is based on the CYC common sense knowledge base [57]. As of 2004, this system could recognize over 30,000 different types of named entities, including many specializations of the handful of types that other systems could recognize. "For example, in addition to a class such as 'Person' available to current information extraction systems, CYC’s extended type system includes many categories of people, including things like Terrorist”; thus, based on a mention of a bomb in a news article, the system could recognize that a particular Person is in fact a Terrorist [28]. The CYC knowledge base has been populated through a combination of “spoon-feeding” of facts by professional researchers, as well as machine learning algorithms that learn new facts by sifting through web pages written by end users. ConceptNet is a second successful attempt to amass information about named entities referenced by natural language text [61]. In order to “teach” this system common sense facts, end users can visit the project’s website and type facts into a web form. As of late 2004, the knowledge base contained facts about 300,000 concepts.

These systems take advantage of contextual features. For example, the words “worked for” often precedes a company name or a person name, so the words “worked for” provide a very good starting point for inferring the data category of the following few words. However, these contextual cues are unavailable or not useful in the problems encountered by end users during the studies described above. For example, if an end-user programmer places a “Phone:” label beside a web form field (so users know what data belongs there), then that label provides no information about whether a particular input is valid. Although these limitations apparently prevent using the information extraction algorithms for input validation, the notion of a named entity was one source of inspiration for the notion of topes, as described by Section 1.2.

2.7.10 Microformats and the semantic web

Microformats are a labeling scheme where a web developer affixes a “class” attribute to an HTML element to specify a category for the element’s text [47]. For example, the developer might label an element with “tel” if it contains a phone number. Commonly-recognized microformat labels are published on a wiki (http://microformats.org/wiki/). Conceptually, microformats are a simplified version of the approach used in the semantic web, where programmers affix complex RDF labels to HTML and XML elements, thereby indicating the category of data that the element contains [9].

Regardless of whether data are tagged with microformat labels or with semantic web labels, people can write programs that download HTML or XML files and retrieve phone numbers or other data values from labeled elements. The values would then require validation and, perhaps, reformatting to a common format. Thus, while microformats and the semantic web involve the categories of data that appear in end-user tasks, these approaches do not actually directly solve the validation and reformatting problems that end users encounter. Another approach is needed.

Chapter 3. Topes

The empirical studies described in Chapter 2 revealed that many user tasks involve validating and reformatting short human-readable strings drawn from data categories such as person names, phone numbers, and university project numbers. These categories often include values that are questionable yet nonetheless valid, and their values typically can appear in multiple formats. The categories are “application-agnostic”, in that the rules for validating and reformatting instances are not specific to a particular application (such as Excel). Rather, the rules are implicitly or explicitly agreed upon by society at large or within an organization.

A tope is an application-independent abstraction that describes how to validate and reformat instances of one such data category. Each is called a “tope”, the Greek word for “place”, because its data category has a *place* in the problem domain. For example, problem domains involve phone numbers and salaries, rather than strings and floats, and it is the user context that governs whether a string is valid. Every phone number is a valid string, but not every string is a valid phone number.

Humans have conceptual patterns for these data categories. The purpose of a tope is to take what people know about a particular category and capture this knowledge in a form that the computer can use. In particular, tope captures two kinds of conceptual patterns, each represented as a mathematical function. First, a tope contains functions describing how to recognize instances of each format on a non-binary scale of validity. For example, a phone number tope might contain a conceptual pattern describing how to recognize strings formatted like “800-987-6543”, with the stipulation that a string is highly questionable if its exchange is “555” (since this exchange is extremely rare in valid phone numbers). Second, if a tope describes a multi-format data category, then it typically also contains conceptual patterns describing how to transform instances from one format to another. For example, phone numbers can be transformed from a format like “800-987-6543” to a format like “800.987.6543” by following a simple rule of changing all hyphens to periods.

In short, a tope captures validation and reformatting rules that describe a data category. It does not explicitly capture *other* information about instances of the data category (such as how often people encounter instances of the category in real life). The functional interface exposed by a tope are precisely intended to help automate validation and reformatting tasks—no more, no less. Thus*,* a tope is a parsimonious computational model of a data category used by people in everyday life.

This chapter begins by laying out the rationale for abstractions describing categories of string-like data. It precisely defines and discusses what topes are, then explores several common kinds of topes. The chapter closes by considering the importance of providing good support for helping users to create correct tope implementations.

3.1 Abstractions for data categories

Each tope is an abstraction that models one category of string-like data. "Abstraction" is used in the following sense: "The essence of abstraction is recognizing a pattern, naming and defining it, analyzing it, finding ways to specify it, and providing some way to invoke the pattern." [114] Here, "pattern" has the dictionary definition of "a consistent or characteristic arrangement" [92] rather than the specific technical meaning of a Design Pattern [36].

Creating an abstraction for a data category involves recognizing the conceptual patterns implicit in that category's valid values, naming the pattern, and carefully defining the pattern. Ultimately, invoking the pattern's operations requires implementing the abstraction. Just as there may be several possible abstractions each describing a particular data category, there may be several different approaches for implementing each abstraction.

For example, consider a simple data category, email addresses. A valid email address is a username, followed by an "@" symbol and a hostname. At the simplest level, the username and hostname can contain alphanumeric characters, periods, underscores, and certain other characters. This abstraction—a pattern that succinctly describes the category—could be implemented in various ways, such as a regular expression, a context-free grammar, or a Java program.

More sophisticated abstractions for email addresses are possible. For example, a more accurate pattern would note that neither the username nor the hostname can contain two adjacent periods. A still more accurate abstraction would note that the hostname usually ends with ".com", ".edu", country codes, or certain other strings.

Some data categories encountered by end users are most easily described in terms of *several* patterns. For example, companies can be referenced by common name, formal name, or stock ticker symbol. The common names are typically one to three words, sometimes containing apostrophes, ampersands, or hyphens. The formal names may be somewhat longer, though rarely more than 100 characters, and they sometimes contain periods, commas, and certain other characters. The ticker symbols are one to five uppercase letters, sometimes followed by a period and one or two letters; more precisely, these symbols are drawn from a finite set of officially registered symbols. These three patterns *together* comprise an abstraction describing how to recognize companies.

In addition to patterns for recognizing instances of a multi-format data category, each tope captures patterns for reformatting instances from one format to another. Again, the word "pattern" implies that certain rules generally describe how to reformat instances of a category. In other words, it is not as if every string in the category has its own particular rules for reformatting; instead, certain rules generally describe all (or most) of the category's instances. For example, instances of the company data category can be reformatted from a ticker symbol format to a full company name by referencing a lookup table. The lookup table embodies a conceptual pattern for how to reformat the strings. As with patterns for recognizing instances, this lookup table pattern could be implemented with a C# "switch" statement, a set of "if/then" conditional statements, a .NET HashMap, or other code constructs. In practice, users create tope implementations through the Tope Development Environment (TDE), as described by Chapter 4, though the tope model is more general and would allow other possible implementation approaches.

3.2 The topes model

The purpose of topes is to validate and reformat strings. Thus, topes capture users' conceptual patterns as mathematical functions involving strings. Each string is a sequence of symbols drawn from a finite alphabet $\sum$, so each string $s \in \sum^*$. A data category D is a set of valid strings.

As shown in Figure 1, each tope τ is a directed graph (F, T). In this graph, a function is associated with each vertex and edge. For each vertex $f \in F$, called a "format", $\texttt{isa}_f$ is a fuzzy set membership function $\texttt{isa}_f : \sum^* \rightarrow [0,1]$. For each edge $(x,y) \in T$, called a "transformation", $\texttt{trf}_{xy}$ is a function $\texttt{isa}_{xy} : \sum^* \rightarrow \sum^*$. Each vertex's `isa` function, also known as a "format validation function", recognizes instances of a conceptual pattern. Each edge's `trf` function, also known as a "transformation function" converts strings between formats.

For a given format vertex f, each $\texttt{isa}_f$ defines a fuzzy set [125], in that $\texttt{isa}_f(s)$ indicates the degree to which `s` is an instance of that format's conceptual pattern. For example, when recognizing strings as company common names based whether they have one to three words, a string `s` with one word would match, so $\texttt{isa}_f(s) = 1$. The conceptual pattern might allow an input `s'` with four words, but with a low degree of membership, such as $\texttt{isa}_f(\texttt{s'}) = 0.1$. The pattern might completely disallow an input `s''` with 30 words, so $\texttt{isa}_f(\texttt{s''}) = 0$.

For each tope $\tau = (F, T)$, the "tope validation function" is defined as

$$\phi(s) = \max_{f \in F}[\texttt{isa}_f(s)]$$

If $\phi(s) = 1$, then τ labels `s` as valid. If $\phi(\texttt{s}) = 0$, then τ labels s as invalid. If $0 < \phi(\texttt{s}) < 1$, then τ labels s as questionable, meaning that it might or might not be valid. The ultimate goal of validation is to identify invalid data. For this particular purpose, τ would be perfectly accurate if $\forall\ \texttt{s} \in \sum^*$, $(\phi(\texttt{s}) = 0) \Leftrightarrow (\texttt{s} \notin D)$.

Each $\texttt{trf}_{xy}$ converts strings from format `x` into equivalent strings in format `y`. For example, as discussed by Section 3.1, a tope implementation for company names might use lookup tables to match common names and official titles with stock symbols. Specifically, for data categories referencing named entities, two strings with different character sequences can refer to the same entity. For example, "Google", "GOOG", and "Google Inc." refer to the same entity, though the three strings are written in different formats. For categories that do not reference named entities, two strings with different character sequences may be equivalent according to that category's underlying semantics. For example, a temperature of "212° F" is equivalent to "100° C".

Transformations can be chained, so in practice, topes do not need to be complete (each vertex directly joined to every other vertex). As a result, the number of functions can grow linearly with respect to the number of formats. Some topes may have a central format connected to each other format in a star-like structure, which would be appropriate when one format can be identified as a canonical format. Alternate topologies are appropriate in other cases, such as when a tope is conveniently expressed as a chain of sequentially refined formats.

Ambiguity can affect the use of transformations in two ways. First, when chaining transformation functions in order to traverse the tope graph from format `x` to format `y`, there may be more than one feasible route through the graph, and each route might produce a different result. A second complication related to choosing a route relates to picking a starting point for the route: to transform a string to format `y`, it is necessary to first determine the starting format `x` of the string, and in some cases, it might be the case that more than one format matches the string. Section 6.1.2 explains how these two practical considerations impact the design of the TDE.

3.3 The interplay between validation and transformation

One crucial design decision reflected in this model is to require each transformation function to accept *any* string, even a string that is not valid according to the source format.

The reason for this decision is that there are many imperfect strings that still need to be reformatted. For example, an `isa` function might assign a score of 0.9 to the string "3043048 Main Street", as the street number has a rather high number of digits. Yet there is no real reason why it could not be transformed to another format, such as "3043048 MAIN ST." In fact, it might even be desirable to reformat a clearly invalid string. For example, "(911) 818-1010" is not a valid phone number, as area codes can never end with "11". Yet ask an end-user programmer to transform it to his favorite format, and he will probably produce "911-818-1010" with very little effort. So a user's conceptual pattern for transforming strings often "works" on many imperfect strings, which is logical because the purpose of transformation is to transform, not to validate.

Accordingly, the job of a `trf` function is to make a "best effort" at transforming rather than to validate (which is properly the job of an `isa` function). Yet the decision to require only that `trf` functions make a "best effort" has a number of consequences. One consequence is that the outputs of transformations might be invalid. Another is that it can sometimes be impossible to recover a string after transforming it. The topes model does not provide any direct support for coping with these consequences, but tools that rely on the topes model can cope with these consequences in a straightforward manner.

Transformations might not produce valid outputs sometimes

By requiring each `trf` function to accept any string, it becomes necessary to accept that each `trf` function may sometimes be unable to produce a valid output. For example, there is no way to transform a meaningless string like "a8`6" to any phone number format, and a `trf` function might output any string at all when faced with that input.

In fact, a `trf` function might produce an invalid string even when it can make sense of the input. Carrying along the example above, a good phone number `trf` might be able to convert "(911) 818-1010" to "911-818-1010", yet the output would still be invalid. Moreover, an output could be questionable or invalid even if the input was perfectly valid. As an example, the person name "Smith, Malcolm" cannot be reliably transformed to any format that includes a middle initial. Even though the input is a perfectly valid instance of a very commonly used person name format, it does not contain the information needed to produce a valid instance of a format that requires a middle initial. For some kinds of data, the `trf` might contain special rules to populate missing fields with default values, but there really is no good default for a person's middle initial.

Conversely, a `trf` function might be so robust that it can take a questionable input and produce a valid output. For example, the string "SmitH, Malcolm" has an unusual mixture of capitalization, so a good `isa` function might notice this and identify the string as questionable. Yet it would be very easy to implement a `trf` function for transforming strings from this format to a format that is all-uppercase with the given name followed by the family name. This function would probably produce a valid string like "MALCOLM SMITH".

In summary, regardless of whether an input is valid, questionable or invalid, the output of a `trf` function could be valid, questionable or invalid.

People manually apply transformations like these every day. In doing so, they sometimes improve the quality of data, and they sometimes reduce the quality of data. This is a result of the structure and rules of the data categories that people use. Modeling and ultimately automating the data categories, including their transformations, does not change this fact of reality. If a model is to be true to reality, then it must faithfully reflect the strengths as well as the weaknesses of that reality. Identifying those weaknesses exposes them so that we as researchers can help people cope with them.

In the context of topes, there is a very straightforward way to help people cope with the fact that transformations might damage the quality of data: code that calls a `trf` function should validate the output using the `isa` function of the target format. This will immediately indicate whether the result is valid, so that a user can be alerted to double-check the output if needed. The key is to use transformation functions for what they are good at—transforming—and rely on format validation functions for validating.

It might be impossible to automatically recover a string after transforming it

Moreover, `trf` functions might destroy information, in the sense that the output of a transformation might not contain enough information to recover the string that was originally transformed. For example, transforming “Malcolm T. Smith” to “Smith, Malcolm” destroys information, making it impossible to transform the string back to a valid instance of the original format.

By definition, a transformation $\mathtt{trf}_{xy}$ is non-invertible any time that there exists some string $\mathtt{s}$ for which $\mathtt{trf}_{yx}(\mathtt{trf}_{xy}(\mathtt{s})) \neq \mathtt{s}$. As function $\mathtt{f}$ is invertible if and only if it is an injection ($\forall$ $\mathtt{s}$, $\mathtt{s}'$ such that $\mathtt{s} \neq \mathtt{s}'$, $\mathtt{f(s)} \neq \mathtt{f(s')}$) and if it is a surjection ($\forall$ $\mathtt{t}$ in the codomain of $\mathtt{f}$, $\exists$ $\mathtt{s}$ such that $\mathtt{f(s)} = \mathtt{t}$). Thus, a tope transformation function $\mathtt{trf}_{xy}$ is non-invertible if it violates the injection property (there are two different strings that map to the same transformation output) or if it violates the surjection property (there is some string that cannot be generated from any string in the source format).

A careful consideration shows that of these two ways for a transformation function to be non-invertible, only violations of the injection property can destroy information—violations of the surjection property do not destroy information.

When the injection property is violated, it can be impossible to look at a string in the target format and determine which of several strings could have possibly generated the output. The discussion above has already identified one way to violate the injection property, by dropping part of a string during a transformation. Another way to violate the injection property is by rounding a number, and there are surely other ways.

On the other hand, if injection is preserved but the surjection property is violated, then there is some string `t` nominally in the target format `y` but which has no equivalent in the source format x. Yet this is irrelevant for the purposes of inverting the transformation of a string `s`: by construction, the output $\mathtt{trf}_{xy}\mathtt{(s)}$ can never be one of those strings that cannot be generated by $\mathtt{trf}_{xy}$ from some string in the source format. Thus, the violation of surjection does not prevent recovering a transformed string through an inverse transformation.

The straightforward way to help users with situations like these is for a tool to retain a hidden copy of a string somewhere before calling transformation functions, then providing convenient "undo" support for recovering the original string. The TDE does not presently happen to provide such support, but it could be added in the future, as "undo" functionality is fairly well understood and easily implemented with a Memento pattern (which succinctly captures information required for restoring a computational element to a previous state) [36].

In fact, the Memento pattern might also be useful when strings are transformed in one tool and then exported to another tool, and the user might want to recover the original strings. In this case, the second tool will not have an "undo" history in memory for reversing the transformation, but the data could be annotated with meta-data containing a Memento when the data are exported from the first tool. That way, the second tool would have the information required for recovering the original strings.

3.4 Common kinds of topes

Four specific kinds of patterns describe most topes.

Enumeration patterns

Values in many data categories refer to named entities [4][26][67], of which only a finite number actually exist. Each such category could be described as a tope with a single format whose membership function $\mathtt{isa}_f(s) = 1$ if $s \in \Delta$, where Δ is a set of names of existing entities, and $\texttt{isa}_f$(s) = 0 otherwise. If two strings can refer to the same named entity, it may be convenient to partition this set, creating multiple formats, each of which recognizes values in a partition. The `trf` functions could use a lookup table to match strings in different formats.

For example, consider a tope for Canada's provinces. The tope might have two formats, one for the full name and one for the postal abbreviation.

An `isa` for the first format `x` could assign 1 for each of Canada's ten provinces:

$\texttt{isa}_x$(s) = 1 if s∈ {"Alberta", "British Columbia",…}, 0 otherwise

Depending on the programmer's purpose for this category, a slightly more sophisticated format could also recognize the three territories of Canada with uncertainty, perhaps with

$\texttt{isa}_x$(s) = 0.1 if s∈ {"Northwest Territories", "Nunavut", "Yukon"}, otherwise

The tope could include a second format `y` that recognizes standard postal abbreviations:

$\texttt{isa}_y$(s) = 1 if s∈ {"AB", "BC"…}, 0.1 if s∈ {"NT", "NU", "YT"}, 0 otherwise

The `trf` functions could use a lookup table to map values from one format to the other.

Similar topes could be used to describe many other data categories, including states in the United States, countries of the world, months of the year, products in a company's catalog, and genders. Each of these data categories contains a small number of values.

List-of-words patterns

Some categories have a finite list of valid values but are infeasible to enumerate, either because the list is too large, too rapidly changing, or too decentralized in that each person might be unaware of valid values known to other people. Examples include book titles and company names. Though a tope could contain a format `x` with $\texttt{isa}_x$(s) = 1 if s∈ Δ (an enumeration pattern), this format would omit valid values.

Most such data categories contain values that are lists of words delimited by spaces and punctuation. Valid values typically must have a limited length (such as one or two words of several characters each), and they may contain only certain punctuation or digits. Typical lengths and characters vary by category.

A tope could contain a format y with $\texttt{isa}_y(\texttt{s}) = r \in (0,1)$ if s has an appropriate length and contains appropriate characters. This could be implemented by splitting the string on word boundaries, then checking constraints on the words. For a tope with an enumeration format x and a list-of-words format y, $\phi(s) = \texttt{isa}_x(\texttt{s}) = 1$ if $\texttt{s} \in \Delta$, otherwise $\phi(s) = \texttt{isa}_y(\texttt{s}) = r$ if s matches the list-of-words format's pattern, and 0 otherwise.

The tope could omit trf functions connecting formats x and y. However, from an implementation standpoint, the list-of-words format could be generated automatically by examining values in Δ and inferring a pattern using an algorithm such as one provided by the TDE (as described by Section 4.4). Conversely, at runtime, if a string s appears often, then it could be added automatically or semi-automatically to Δ.

Numeric patterns

Some data categories contain string representations of numeric data that conform to constraints. Examples include salaries, weather temperatures, and bicycle tire sizes. In many cases, a dimensional unit is present or otherwise implicit, with possible unit conversions. For example, "68° F" is equivalent to "20° C". Weather temperatures less than "-30° F" or greater than "120° F" are unlikely to be valid.

A tope having one format for each possible unit could describe such a category. For each format f, $\texttt{isa}_f(\texttt{s}) = 1$ if s contains a numeric part followed by a certain unit such as "° F", and if the numeric part meets an arithmetic constraint. If s comes close to meeting the numeric constraint, then $\texttt{isa}(\texttt{s}) = r \in (0,1)$.

For these data categories, converting values between formats usually involves a linear transform. Thus, each $\texttt{trf}_{xy}$ could multiply the string's numeric value by a constant and add an offset, then change the unit label to the unit label required for the destination format y.

Some data categories are numeric but have an implicit unit or no unit at all. An example is a North American area code, which must be between 200 and 999, and which cannot end with 11. For such categories, a suitable tope could have one format.

The choice to treat data as strings in the topes model can induce a certain amount of information loss during numeric operations (for example, due to fixed-precision rounding). As discussed by Section 3.3, this can lead to some non-invertibility of transformations. Topes might not be a precise enough for use in high-precision computations (for which one of the analysis frameworks mentioned by Section 2.7.2 might be more appropriate), but high precision is not required for the kinds of user tasks (discussed by Sections 2.3 and 2.4) that motivated the development of the topes model.

Hierarchical patterns

Values in many data categories contain substrings drawn from other data categories. One example is American mailing addresses, which contain an address line, a city, a state, and a zip code. An address line such as "1000 N. Main St. NW, Apt. 110" is also hierarchical, containing a street number (recognizable with a numeric pattern), a predirectional (an enumeration pattern), a street name (a list-of-words pattern), a street type (an enumeration pattern), a postdirectional, a secondary unit designator (an enumeration pattern), and a secondary unit (a numeric pattern). Some parts are optional. In most cases, spaces, commas, or other punctuation serve as separators between adjacent parts. Other examples of hierarchical data include phone numbers, dates, email addresses, and URLs.

A format `x` for a hierarchical pattern could constrain each of its parts to match a corresponding tope. It could also introduce "linking constraints" that operate on the string as a whole to require that the parts are properly coordinated. For example, the format's `isa` function could evaluate the conjunction of all these constraints as follows:

$$\mathtt{isa}_{\mathtt{x}}(s) = \prod_i \phi_i(s_i) \prod_j \chi_j(s)$$

Here, i ranges over the value's parts, each s_i is the substring for part i (which should be valid according to tope τ_i), and $\phi_i(s_i)$ is the tope validation function for τ_i; j ranges over "linking constraints" that involve multiple parts, and $\chi_j(s) \in [0,1]$ applies a penalty if any linking constraint is violated. As an example of a linking constraint, in a date, the set of allowable days depends on the month and the year. If `s` violates this constraint, $\mathtt{isa}_{\mathtt{x}}(s) = \chi_j(s) = 0$. The `isa` function could be implemented by parsing the string with a context-free grammar, then testing nodes in the tree against the constraints.

The `isa` function above uses the most straightforward approach for evaluating the conjunction of constraints, and in the context of fuzzy sets it is referred to as an "algebraic product" [125]. Multiplication is a natural extension of probabilistic conjunction, but the semantics of fuzzy membership functions are different than that of probabilities: there is no requirement that the resulting `isa` function must return a probability. If it happens to return a probability, that is fine. But it need not return a probability. Even if the ϕ_i

and χ_j constraints are not statistically independent, their product still yields a valid isa score (as each ϕ_i and χ_j returns a value between 0 and 1, so their product is in the same range).

The main potential problem with using multiplication to evaluate the conjunction is that isa scores will quickly tend toward zero when strings violate multiple constraints, which may not accurately reflect a user's conceptual pattern for the format. Another common approach for evaluating conjunctions is simply to take the minimum among the fuzzy constraints [125]. This would yield

$$\texttt{isa}_x(s) = \min(\phi_1(s_1), \phi_2(s_2), \ldots, \phi_{\#parts}(s_{\#parts}), \chi_1(s), \chi_2(s), \ldots, \chi_{\#link-constr}(s))$$

This approach has the opposite potential problem of the one for multiplication: it will not penalize strings for violating multiple constraints, which might not match a user's conceptual pattern. Consider a questionable street address like "999787 Elm UPAS", which has a slightly long street number and an unusual but valid street type (the officially sanctioned US Postal Service abbreviation for "Underpass"). When using the minimal-value approach, this string would be considered just as questionable as "999787 Elm Street". While this equality might sit well with some people, it might not match the conceptual pattern of other people. Thus, different people might prefer to take different approaches for implementing this tope. The topes model allows either approach or any other approach for computing isa scores between 0 and 1.

3.5 Consistency between a tope and a tope implementation

The preceding section has presented common kinds of topes and outlined approaches for implementing topes. But having implemented a tope, what if the tope implementation is not correct? Put another way, a user might be thinking of a certain conceptual pattern, then use a tool suite to implement that pattern. What if the tope implementation is inconsistent with the intended abstraction?

One approach would be for the user to create a formal specification describing the abstraction. For example, the specification could formally state that the tope should have three isa functions for matching a certain set of three formats, as well as four trf functions to connect the formats in a certain pattern (for example, as shown in Figure 1). The specification could describe additional details that make this tope distinct from other topes. For example, if the tope was for recognizing phone numbers, the specification could state that any string accepted by any format validation function should contain exactly 10 digits. Finally, (hypothetically speaking) the user could provide this specification along with a particular tope implementation to a model checker that would evaluate whether the implementation matched the specification. In some areas of professional programming, formal approaches like this one have achieved a certain amount of noteworthy success. For example, NASA has used the Java Pathfinder model checker to find defects in real spacecraft [22]. However, this approach does not guarantee that the implementation matches the abstraction. One obvious reason is that the model checker can only check properties that are Turing-decideable; in particular, some very simple tope implementation properties (such as requiring that all `isa` functions terminate for all inputs) would be impossible to check.

However, in the setting of topes, the main limitation of this formal approach is that it assumes that the programmer has the time and expertise to write a formal specification. While NASA and organizations with similar high-dependability requirements may sometimes find it appropriate to invest in hiring such a person and giving him time to write a specification, many other companies' projects are judged successful even when code has implementation errors. Consequently, most professional programmers currently rely on less formal methods such as code reviews and testing for detecting programming errors, and they never develop expertise at writing formal specifications. End-user programmers, of course, have even less expertise in formal modeling than professional programmers do. Moreover, their motivations are to get the programming done and then get back to "work", so they are unlikely to invest time in writing formal specifications, even if they had the expertise.

Consequently, informal approaches will be needed to help users detect inconsistencies between topes and tope implementations, or at least to override the outputs of misbehaving tope implementations. After the following chapter gives a high-level overview of the TDE, succeeding chapters will delve into system features provided to help users create correct tope implementations. For example, the system provides a feature so that end-user programmers can enter example strings to test with the tope implementation

(Section 4.3). In addition, the system includes a static analysis for automatically detecting common errors that can occur when users combine two tope implementations into one(Section 7.1.3).

In short, topes themselves do not provide formal mechanisms to prevent users from creating a tope implementation that is inconsistent with the intended conceptual pattern. However, as the next few chapters will show, the simplicity of the model lends itself to various semi-automatic methods for making inconsistencies more obvious and easily correctable, ultimately enabling end users to more rapidly validate and reformat data than is possible with currently-practiced techniques.

Chapter 4. Data descriptions and the Tope Development Environment

Chapter 3 introduced the concept of a tope, which is an abstraction describing how to recognize and reformat strings in one data category. Ultimately, invoking a tope's operations requires implementing them. Chapter 3 mentioned several implementation approaches, including lookup tables and context-free grammars, each appropriate for a different kind of tope.

Some programmers might feel comfortable implementing lookup tables and grammars by coding in a general-purpose textual programming language such as C#. But for most of topes' target user population, a general-purpose textual language would be a poor choice since, as noted by Section 2.1, only a small fraction of all end-user programmers are familiar with such languages.

What, then, is a more appropriate mechanism for helping users to implement topes? As a general starting point, most end users rely on visual languages, such as the diagramming languages in Microsoft Visio or Access. Even when they use Excel, which supports textual formulas, people still much more commonly use it to store and organize strings than to write formulas—that is, they rely on Excel more heavily for its visual layout of information than for its textual programming language. These considerations suggest that a visual language might be a good starting point for helping users to implement topes.

When designing visual languages, the Natural Programming Project team has often found it useful to match the primitives and overall language structure to the terms and concepts used by people to describe programs [77]. This helps to minimize the "gulf of execution" that users must cross when translating their intentions to language constructs [81]. Section 2.5 has already discussed a pilot study that asked administrative assistants to describe instances of commonly-occurring data categories, revealing that users tend to describe data as a series of constrained parts. This provides the basis for designing a visual language for users to describe instances of data categories. From these descriptions, a tool can then automatically implement topes.

To support this approach, the Tope Development Environment (TDE) includes a tool called Toped^{++} that presents a declarative visual language to users who want to implement topes. A "data description" is a program expressed in this visual language. For a given data category, a data description identifies the constrained parts that strings must have in order to be instances of that data category, as well as the ordering of those parts and any intervening separators. The TDE automatically generates tope implementations from data descriptions. For example, someone might use the TDE's Toped^{++} tool to create a data description for American phone numbers; this data description would identify three parts (area code, exchange, and local number), constraints on each part, and the allowed permutations and intervening separators for those parts.

Data descriptions, as well as the tope implementations automatically generated from the data descriptions, are application-agnostic. That is, they make no reference to any particular user application such as Excel. Architecturally, this design represents a separation of concerns: all application-specific functionality is factored into separate TDE modules, called add-ins. For example, the TDE includes an add-in for Microsoft Excel; this add-in takes care of all application-specific minutiae, such as how to read strings from spreadsheet cells, how to display error messages (in tooltips), and so forth. The work of add-ins sounds pedestrian, but they play the vital role of seamlessly integrating tope implementations into the user interfaces of applications, in order to ensure a smooth user experience. Moreover, by taking care of application-specific concerns, add-ins are crucial to maintaining the application-agnosticism of data descriptions and the automatically generated tope implementations. This, in turn, facilitates reuse, since each tope implementation can be called (without modification) from many different applications.

In order to support reuse across multiple users, the TDE includes a repository system where data descriptions can be uploaded and downloaded: a user can download an existing data description from which the TDE generates a tope implementation that is the same as the tope implementation that the original user's TDE generated (provided that both users are running same version of the TDE). In this sense, the TDE supports "reuse" of tope implementations by different people, though technically it is a data description that is being reused.

Because users are likely to have some examples of data that they would like to validate or reformat, the repository includes a "search-by-match" feature whereby users can look for data descriptions whose tope implementations recognize user-specified examples. For example, users could search for data descriptions whose tope implementations that match “5000 Forbes Ave.” This search feature is also available locally, so that the user can search his computer for a needed data description.

In short, the TDE is more than just a tool for creating topes, as it includes software for *using* tope implementations and for *sharing* data descriptions. Traditionally, integrated development environments (IDEs) for professional programmers have offered similar support to help people use and share code. For example, Microsoft Visual Studio provides runtime libraries required to execute programs written in C++, and it is integrated with a Codezone repository for code sharing. Likewise, by supporting use and reuse of data descriptions, the TDE aims to help people gain the maximal benefit from data descriptions.

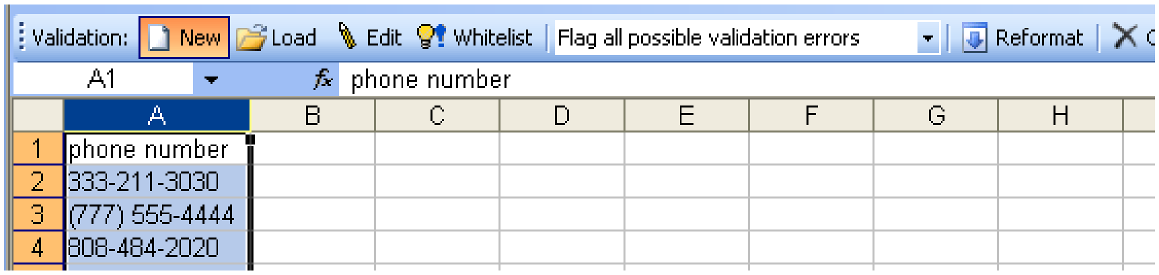
This chapter begins with a walk-through of the steps that a user takes to create a data description and use the TDE-generated tope implementation to validate and reformat strings. It then discusses the data description editor, Toped^{++} and supporting algorithms related to creating data descriptions. Subsequent chapters will describe the internal details of how the TDE uses these data descriptions to generate the format validation and transformation functions of tope implementations, as well as the details of features in the TDE to support reuse of tope implementations.

4.1 Walk-through: Creating and using data descriptions

As described in Chapter 2, users are often faced with tasks that require actions to validate and reformat strings. To illustrate the process of using the TDE to automate these steps, consider a simple scenario where a user has a spreadsheet of phone numbers in various formats. The user needs to validate them and put them into a consistent format.

The TDE’s add-in for Microsoft Excel is a toolbar (Figure 9). This add-in acts as a bridge between the Excel application and the rest of the TDE.

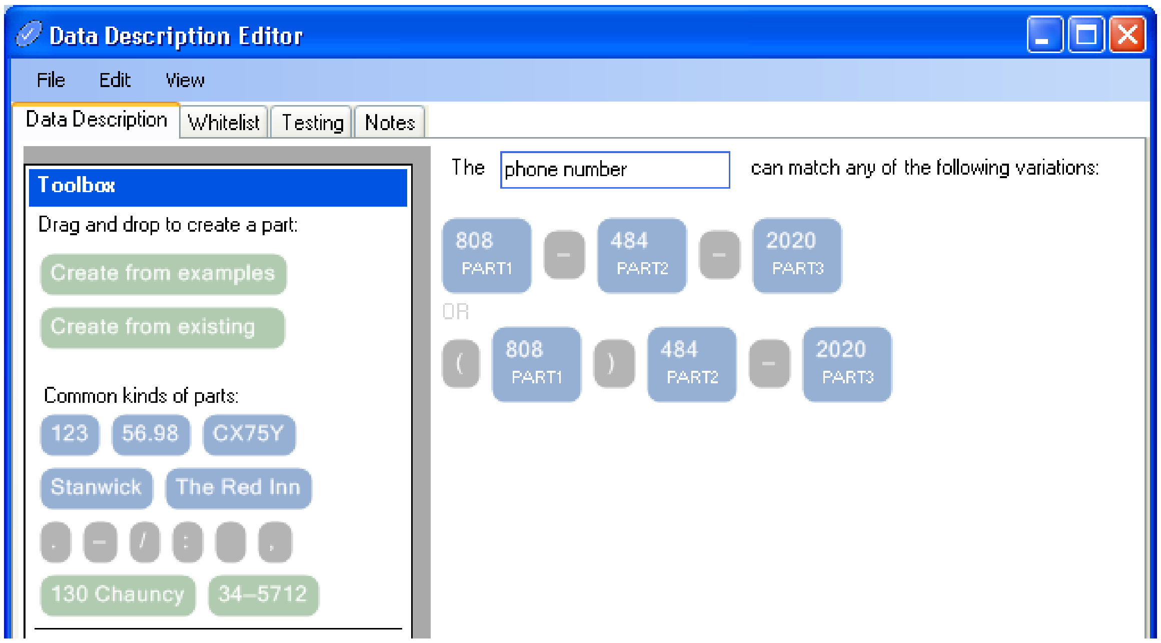
Figure 9. TDE add-in for Microsoft Excel (the "Validation" toolbar) and data to be validated


Validation: New Load Edit Whitelist
Flag all possible validation errors
Reformat
A1
fx phone number

	A	B	C	D	E	F	G	H
1	phone number							
2	333-211-3030							
3	(777) 555-4444							
4	808-484-2020							

To create a new data description, the user highlights the strings to be validated or reformatted and clicks the toolbar's "New" button. The add-in reads these strings and passes them to the TDE, which infers a boilerplate data description. The TDE presents the data description in the Toped++ editing tool for review by the user in a modal dialog window (Figure 10).

Figure 10. An inferred data description with two variations, each containing separators and parts



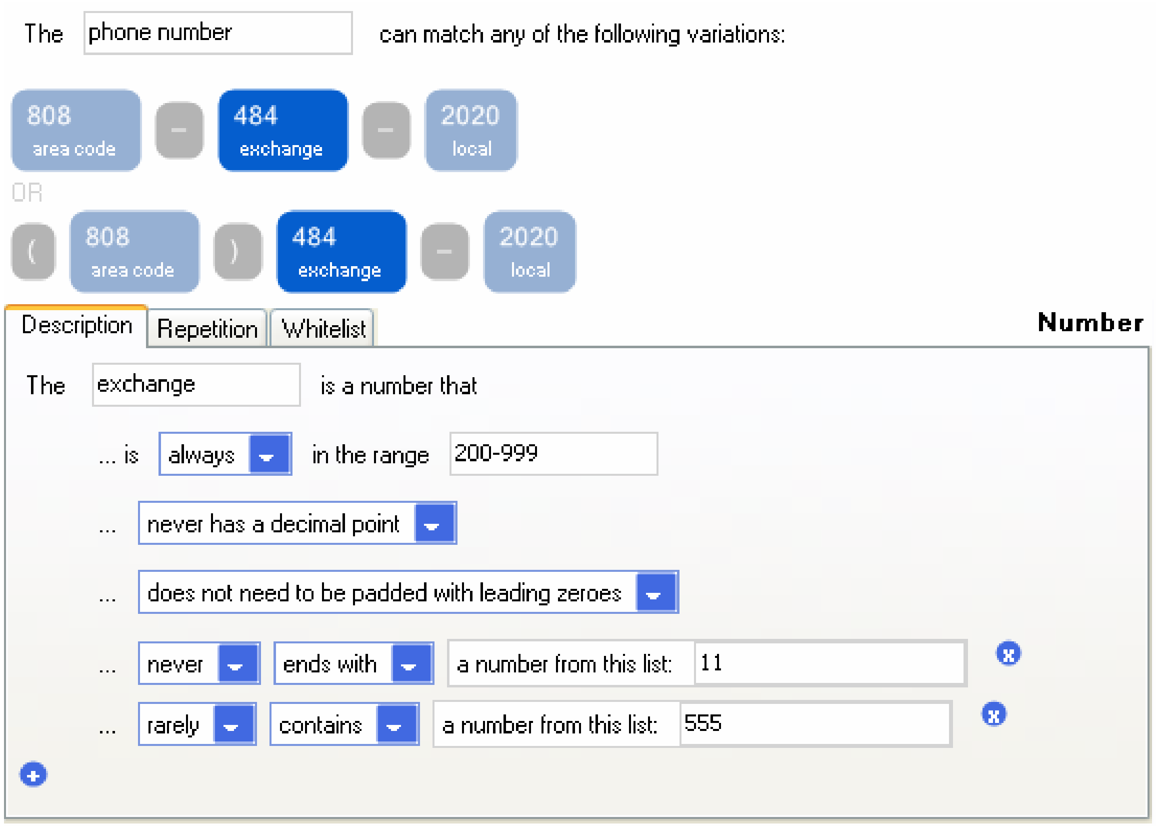
A data description contains one or more variations, each of which Toped++ presents as a row of icons on the screen.[2] Each icon corresponds to a part in the variation or

[2] Toped++ uses the word "Variation" in the sense of "a different form of something; variant" [92]. Perhaps "variant" would have been more precise, but "variation" seemed more desirable because it is probably more familiar to end users than "variant" (as evidenced by nearly twice as many Google hits for "variation" versus "variant").

a separator between parts. Parts can appear in more than one variation. For example, the data description shown in Figure 10 has two variations containing the same three parts but different separators. Each part icon shows the name of the part (which defaults to a name like "PART1" because the system does not know what names are used by humans for the parts) as well as a string that is an example of the part

If the user clicks on a part's icon, Toped^{++} displays a box underneath the variations so that the user can edit the part's example name and its constraints, which might be "soft" in the sense that they are usually satisfied but can sometimes be violated by valid strings. For example, the user can give names to the parts of a phone number and can add constraints indicating that a phone number exchange never ends with "11", and it rarely contains "555" (Figure 11).

Figure 11. Naming the parts of the data description and editing their constraints


The phone number can match any of the following variations:

808 area code - 484 exchange - 2020 local

OR

(808 area code) 484 exchange - 2020 local

The exchange is a number that

... is always in the range 200-999

... never has a decimal point

... does not need to be padded with leading zeroes

... never ends with a number from this list: 11

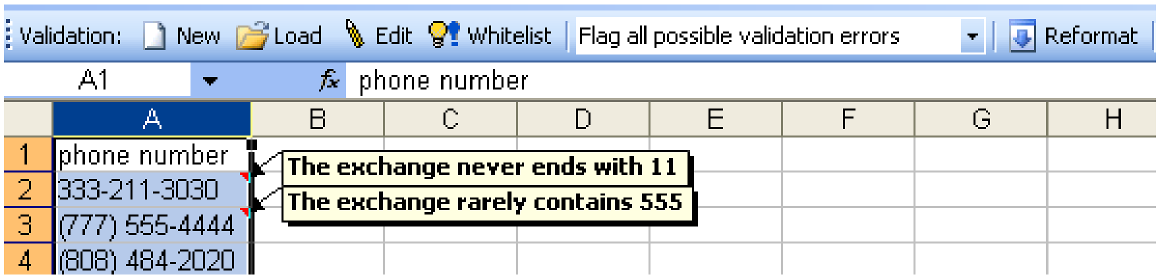
... rarely contains a number from this list: 555

After the user finishes editing the data description, the TDE automatically generates a tope implementation based on the data description. For each highlighted cell, the

TDE's Excel add-in then passes the cell's string through the tope implementation's `isa` functions in order to validate the string. Based on the `isa` functions' return values, the add-in flags each invalid or questionable string with a small red triangle (Figure 12). For each triangle, the add-in also provides a tooltip that briefly explains why the cell is flagged.

The user can tab through these messages using Excel's "Reviewing" functionality, can use the drop-down on the toolbar to filter out messages for questionable strings (thereby only showing tooltips for definitely invalid strings), and can delete any error message to override it. As discussed in detail by Section 4.3, data descriptions can have a "whitelist" indicating strings that should be accepted even though they do not match the data description. To add one or more strings to the whitelist, the user can click the "Whitelist" button in the add-in. From a user interface standpoint, this cancels the corresponding strings' error messages and ensures that those strings are never flagged in the future.

Figure 12. Annotating invalid or questionable cells with red triangles and tooltips



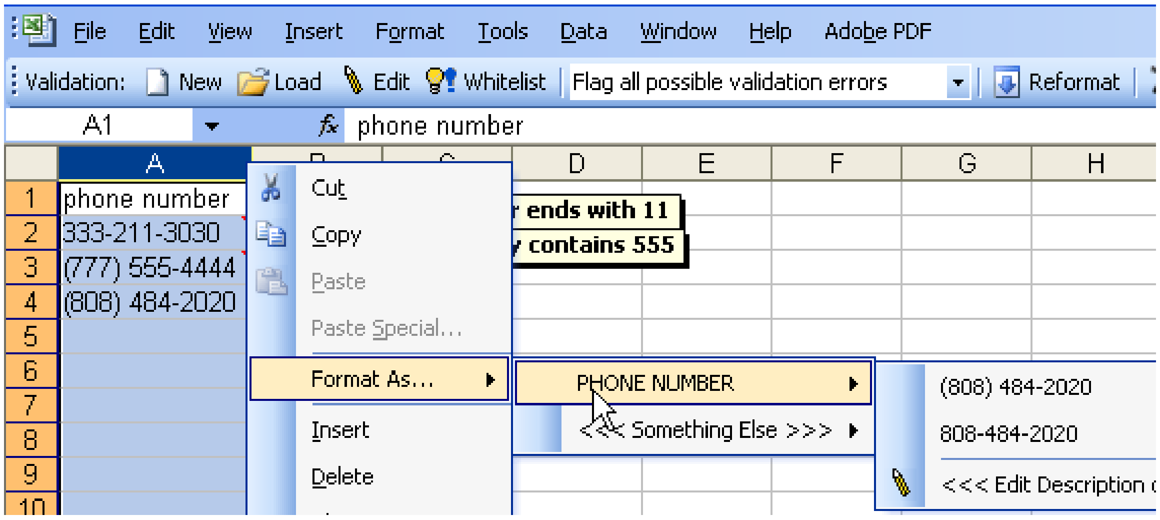
	A	B	C	D	E	F	G	H
1	phone number							
2	333-211-3030							
3	(777) 555-4444							
4	(808) 484-2020							

The exchange never ends with 11

The exchange rarely contains 555

To reformat strings, the user can right-click on the highlighted cells. The add-in detects this action and provides a drop-down menu illustrating what the input strings would look like if reformatted to each available format (Figure 13).

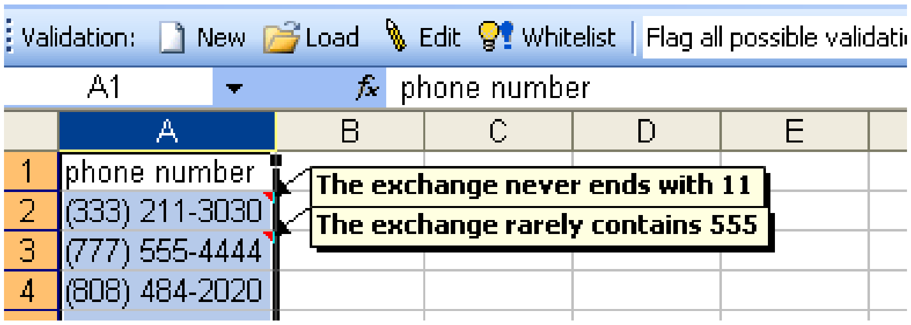
Figure 13. Browsing the formats available for reformatting



	A
1	phone number
2	333-211-3030
3	(777) 555-4444
4	(808) 484-2020

When the user selects one of the options, the add-in reads each string and passes it through the tope implementation's `trf` functions for reformatting to the user's selected format. The add-in displays the result in the spreadsheet (Figure 14). In addition, after reformatting each string, the add-in passes the new string through the `isa` function for the user's selected format, in order to check whether the `trf` function generated a valid output (since, as noted by Section 3.3, `trf` functions may produce invalid or questionable outputs). Finally, the add-in updates each red triangle and tooltip as needed.

Figure 14. Results of reformatting example phone numbers with a tope

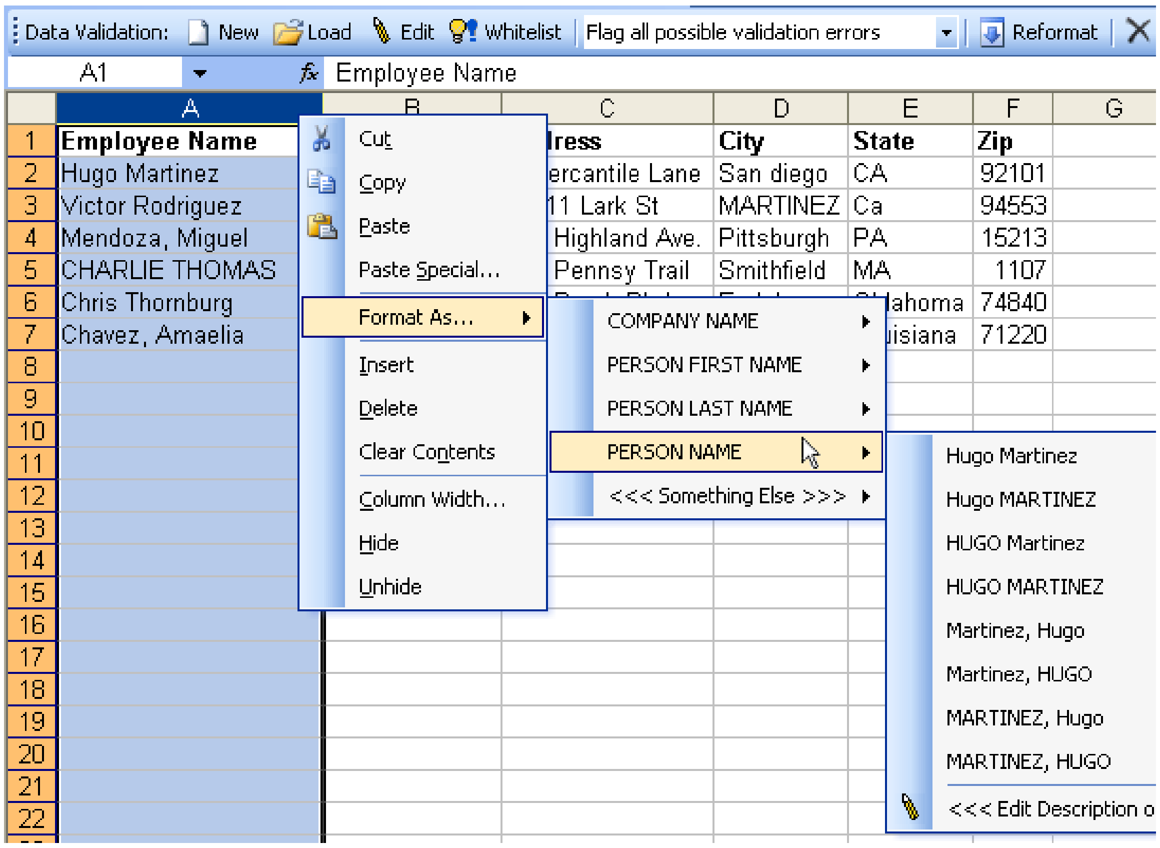


	A
1	phone number
2	(333) 211-3030
3	(777) 555-4444
4	(808) 484-2020

The exchange never ends with 11
The exchange rarely contains 555

Right-clicking to select a data description actually initiates an implicit search for data descriptions on the user's computer whose tope implementations recognize the highlighted cells or whose names include the words in the first row of the spreadsheet. In Figure 13, the TDE is searching for data descriptions whose tope implementations match the three strings "333-211-3030", "(777) 555-4444", and "(808) 484-2020", and whose names include the words “phone” and “number”. Sometimes, more than one data description will match this search query (Figure 15).

Figure 15. Browsing data descriptions recommended based on highlighted cells and heading

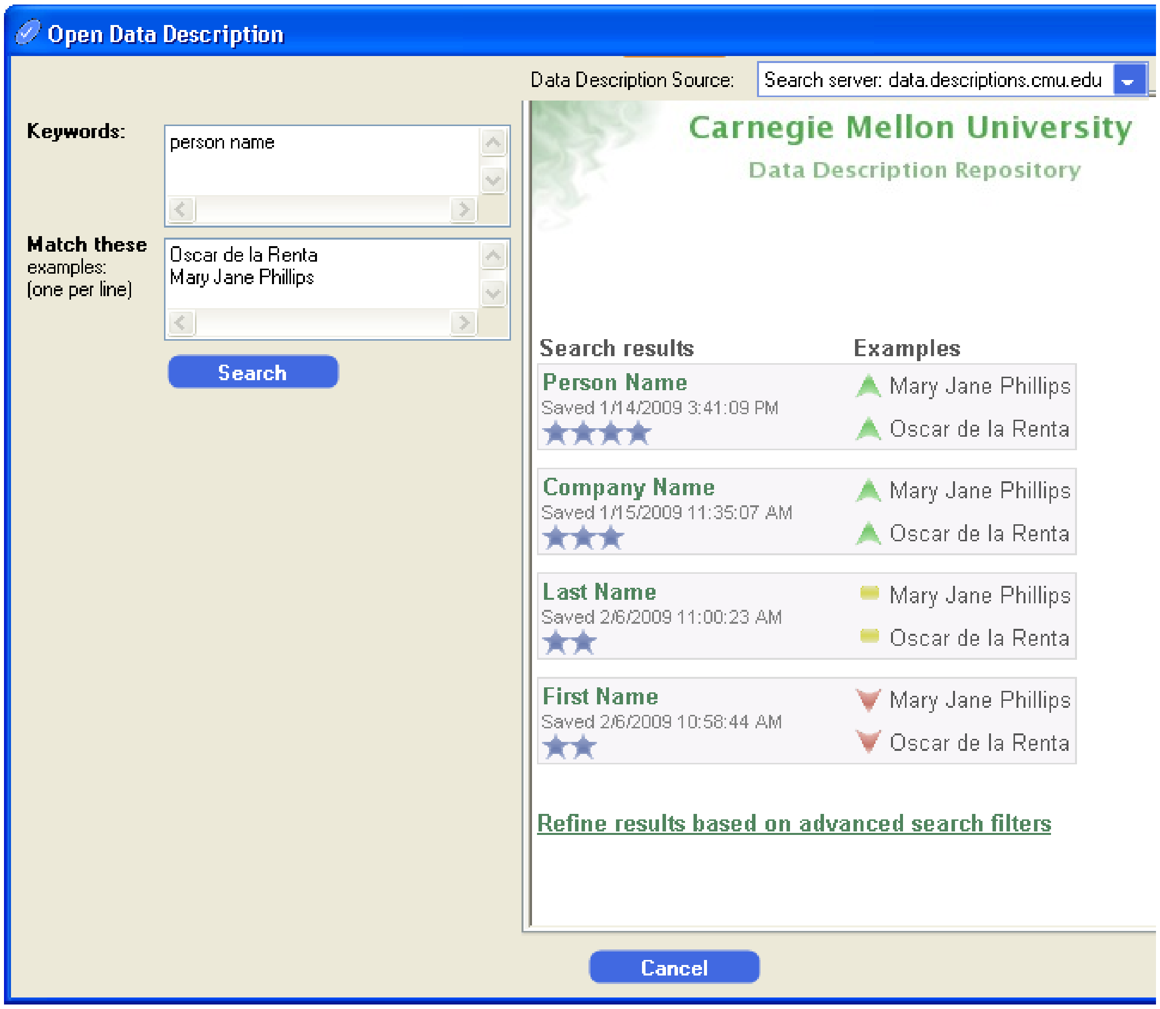


	A	B	C	D	E	F	G
1	Employee Name		ress	City	State	Zip	
2	Hugo Martinez		ercantile Lane	San diego	CA	92101	
3	Victor Rodriguez		11 Lark St	MARTINEZ	Ca	94553	
4	Mendoza, Miguel		Highland Ave.	Pittsburgh	PA	15213	
5	CHARLIE THOMAS		Pennsy Trail	Smithfield	MA	1107	
6	Chris Thornburg				lahoma	74840	
7	Chavez,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

The user can initiate a finer-grained search for a data description using the “Load” button on the toolbar. This brings up a window where the user can specify particular keywords and example strings that should be used to search for a data description. The window also allows the user to browse through all data descriptions stored on the computer.

While the user can reuse data descriptions from the computer, another way to acquire data descriptions is by downloading them from online repositories. The window for viewing remote repositories is actually the same one as the window used for browsing the local computer (Figure 16); to browse the local computer, the user chooses the local computer in the drop-down list at the top of the screen. If the user selects a data description from a remote repository, then the TDE copies the data description from the remote repository to the computer.

Figure 16. Searching the repository to select a data description


Open Data Description

Data Description Source: Search server: data.descriptions.cmu.edu

Keywords: person name

Match these examples: (one per line)
Oscar de la Renta
Mary Jane Phillips

Search

Carnegie Mellon University
Data Description Repository

Search results	Examples
Person Name Saved 1/14/2009 3:41:09 PM ★★★★	▲ Mary Jane Phillips ▲ Oscar de la Renta
Company Name Saved 1/15/2009 11:35:07 AM ★★★	▲ Mary Jane Phillips ▲ Oscar de la Renta
Last Name Saved 2/6/2009 11:00:23 AM ★★	▬ Mary Jane Phillips ▬ Oscar de la Renta
First Name Saved 2/6/2009 10:58:44 AM ★★	▼ Mary Jane Phillips ▼ Oscar de la Renta

Refine results based on advanced search filters

Cancel

While some repositories might be "global", containing data descriptions of interest to a wide range of users, other repositories might be organization-specific and contain only data descriptions of interest to smaller groups of users. For example, the repository software might be installed on a server of the Carnegie Mellon University intranet, so that users can share and download data descriptions for data categories only used by the university. The present repository prototype is "wiki-style" in the sense that any user can upload and edit any data description, but additional authentication controls could easily be put into place to manage user actions. This would allow university system administrators, for example, to maintain control over some university-specific data descriptions.

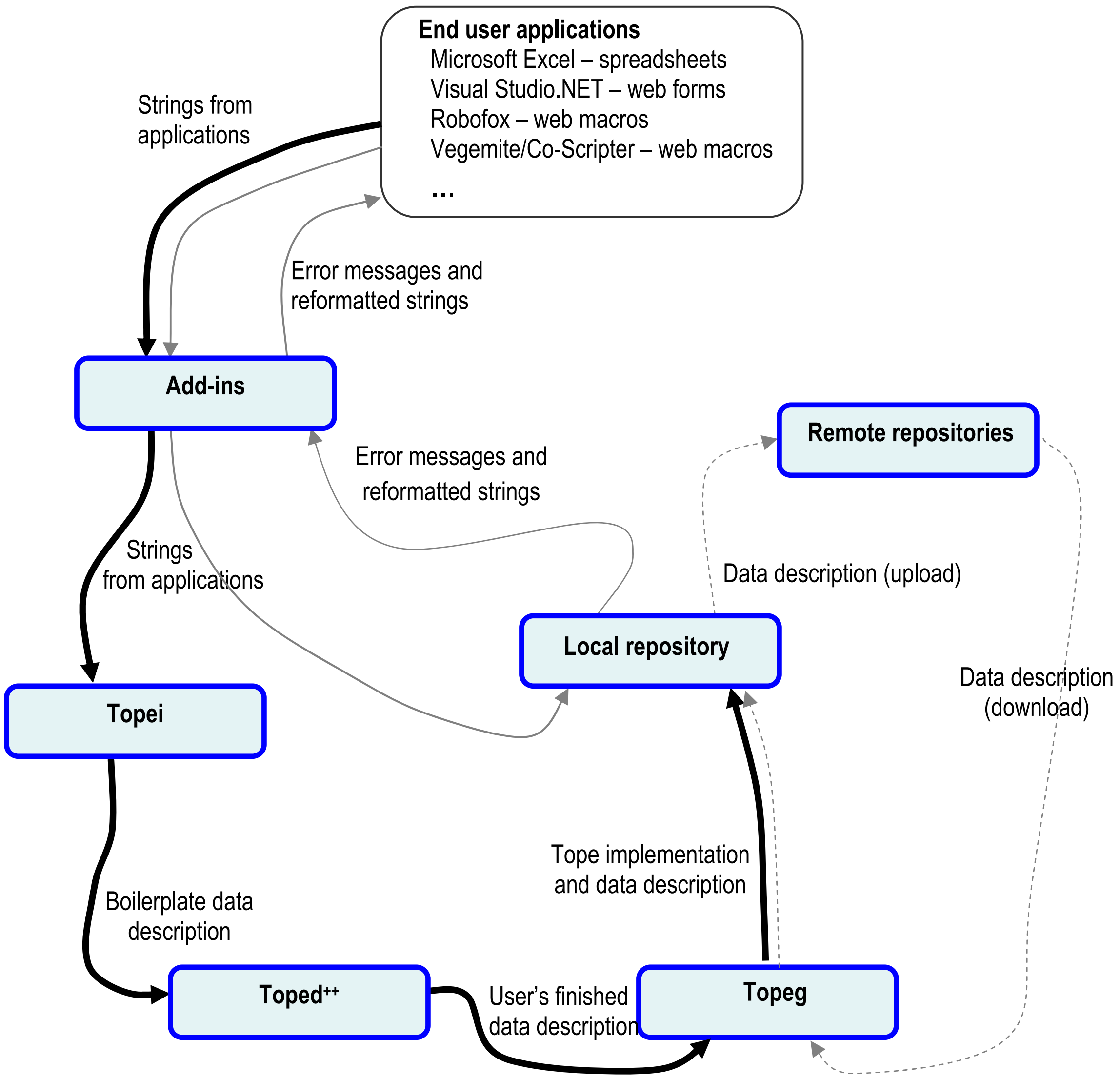
4.2 Mapping steps of the walk-through to elements of the TDE

The walk-through presented by Section 4.1 has illustrated all of the major modules of the TDE, though some have not yet been explicitly mentioned by name. These include the following:

- Add-ins, which are responsible for all application-specific concerns, especially reading strings out of applications as well as displaying error messages and reformatted strings in applications
- Topei, which infers a boilerplate data description from user-provided example strings
- Toped^{++}, which displays data descriptions for review, editing, and testing by users
- Topeg, which generates tope implementations from data descriptions
- Local repository, which stores tope implementations in the local computer's memory and records their respective data descriptions in files; add-ins can pass strings into the tope implementations stored in the local repository
- Remote repositories, which store data descriptions on servers to promote sharing between users (each remote repository internally also has an instance of Topeg, so that it can convert data descriptions into tope implementations as needed in order to service search queries by users)

Figure 17 summarizes these modules and the key data flows among them.

Figure 17. Tope Development Environment (shaded), with data flow indicated by labeled arrows Dark solid lines: data flow as a data description is created and used to implement a tope. Grey solid lines: data flow as strings are validated or reformatted using tope implementations residing in the local repository. Light grey dashed lines: data flow as data descriptions are shared via remote repositories.


End user applications
Microsoft Excel – spreadsheets
Visual Studio.NET – web forms
Robofox – web macros
Vegemite/Co-Scripter – web macros
…
Strings from applications
Error messages and reformatted strings
Add-ins
Remote repositories
Error messages and reformatted strings
Strings from applications
Data description (upload)
Local repository
Data description (download)
Topei
Tope implementation and data description
Boilerplate data description
Toped++
User's finished data description
Topeg

The walk-through of Section 4.1 began by showing Microsoft Excel and the TDE's toolbar add-in for that application. The TDE also includes add-ins for other popular end-user programming tools. For example, the TDE currently provides an add-in for Microsoft Visual Studio.NET; this add-in enables end-user programmers to create web forms that validate strings in textboxes before the forms are submitted to the server. The TDE also currently includes add-ins for Robofox and CoScripter, which are tools for creating web macros to automate a series of interactions with one or more web sites (such as navigating a browser to a web page, copying a string from the page, and then pasting it into a form on another web page). Each add-in is responsible for reading text from an application-appropriate location: The Excel add-in validates strings in spreadsheet cells, the Visual Studio add-in validates strings in web form textboxes, and the web macro tool add-ins validate strings copied by macros from web pages.

The walk-through then showed an inferred data description being edited in Toped^{++}. Section 4.3 describes Toped^{++} in more detail, while Section 4.4 describes the Topei module, which infers boilerplate data descriptions based on examples provided by the add-ins.

When the user had finished editing the data description in the walk-through, the TDE automatically generated a tope implementation. Internally, the `isa` functions test strings against automatically generated context-free grammars, and the `trf` functions reformat strings by parsing them with the grammar and rewriting nodes of the parse trees. Chapters 5 and 6 describe Topeg, the module that generates context-free grammars and tree rewriting rules to create tope implementations.

The walk-through mentioned that the user could search the local and remote repositories in order to reuse an existing data description. The repositories are responsible for storing and indexing data descriptions and tope implementations. The local repository and the remote repository are actually composed of nearly identical software. That is, the repository software used for servers is also running locally on the user's computer. When the Excel add-in recommends data descriptions for reuse (as in Figure 15), it actually is querying this local repository and showing the results in a context-menu. Chapter 7 and Chapter 8 describe the repository software and its search features in detail.

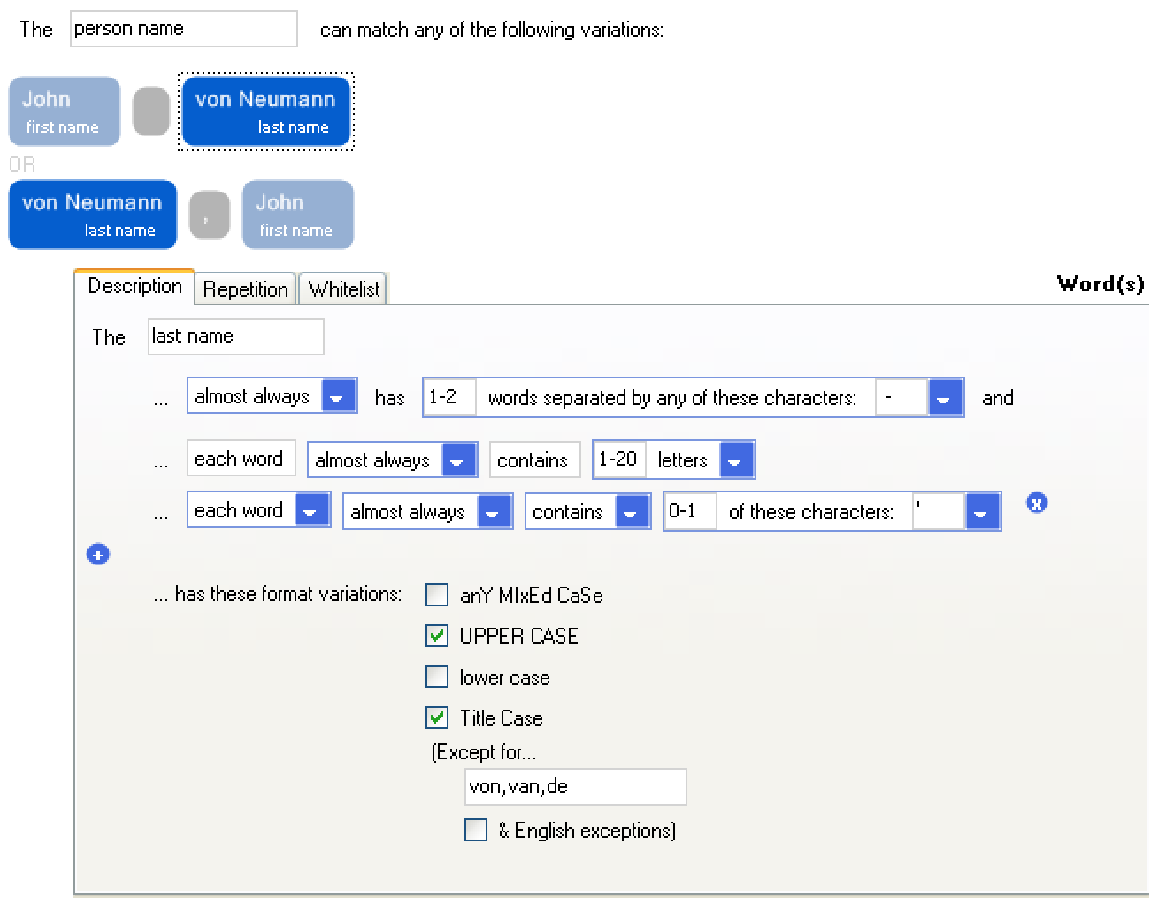
Ultimately, add-ins validate and reformat strings by passing them into the `isa` and `trf` functions of tope implementations residing in the local repository. They update the application's user interface to display error messages and reformatted strings in an application-appropriate way. For example, while the Excel add-in used red triangles and tooltips to show overridable error messages, the Visual Studio.NET add-in uses DHTML to display error messages dynamically on the web page, and (by default) the application user can only override error messages for questionable strings. The reason for this difference in overridability is that the user of a web page is probably a different person than the end-user programmer who created the validation rules for the web page, and programmers typically create validation code for web forms so that other people cannot submit invalid data to the server. Since add-ins take responsibility for all these application-specific concerns, tope implementations can be entirely application-agnostic. Section 7.1.1 delves into this important aspect of reuse in more detail.

4.3 Details of editing data descriptions in Toped^{++}

In accordance with how administrative assistants described instances of common data categories (Section 2.5), Toped^{++} is based on the approach of describing strings as a series of constrained parts. A data description may contain more than one "variation", each of which appears as a row of part icons with separators in between the parts. For example, the phone number data description shown in Figure 11 has three distinct parts, each of which appears in two different variations. As another example, the data description shown in Figure 18 has two distinct parts appearing in two variations. A part's constraints are the same no matter where that part appears. For instance, the area code part in Figure 11 has the same constraints when it appears in the first and second variations. In addition, the data description as a whole can have a JavaScript-coded "linking constraint" that checks relationships between the parts.

Figure 19 shows the internal structure of a data description, in general, while Figure 20 shows the internal structure of the data description that Toped^{++} has displayed for editing in Figure 18.

Figure 18. Editing a person name data description in Toped^{++}.
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. Clicking ⊕ adds a constraint while clicking ⊗ deletes the constraint.



The person name can match any of the following variations:

John first name | von Neumann last name

OR

von Neumann last name | , | John first name

Description | Repetition | Whitelist — Word(s)

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 19. UML class diagram of the internal structure of data descriptions
The kind of a part imposes restrictions on what kinds of constraints the user may add to the part (as discussed by the text below).

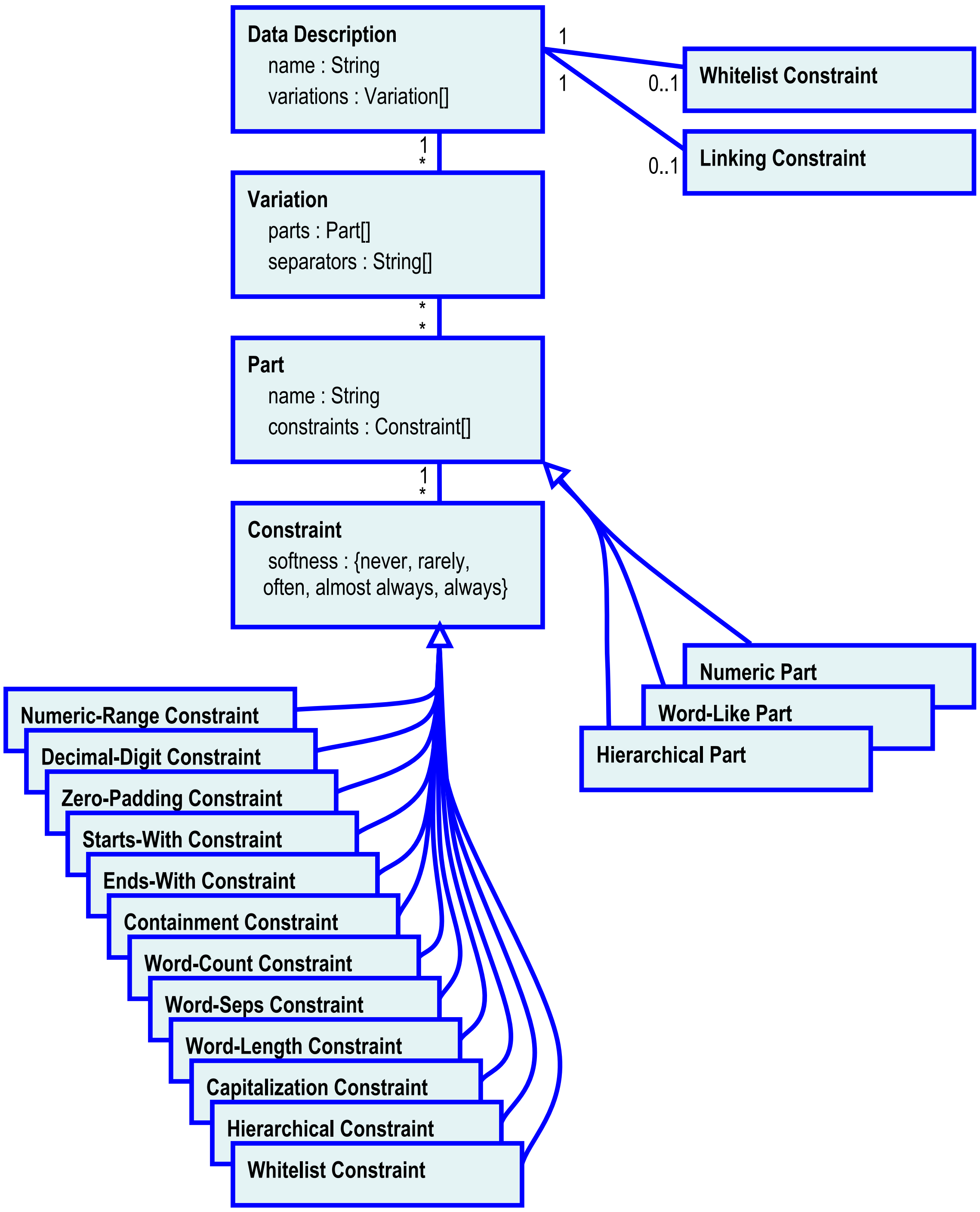
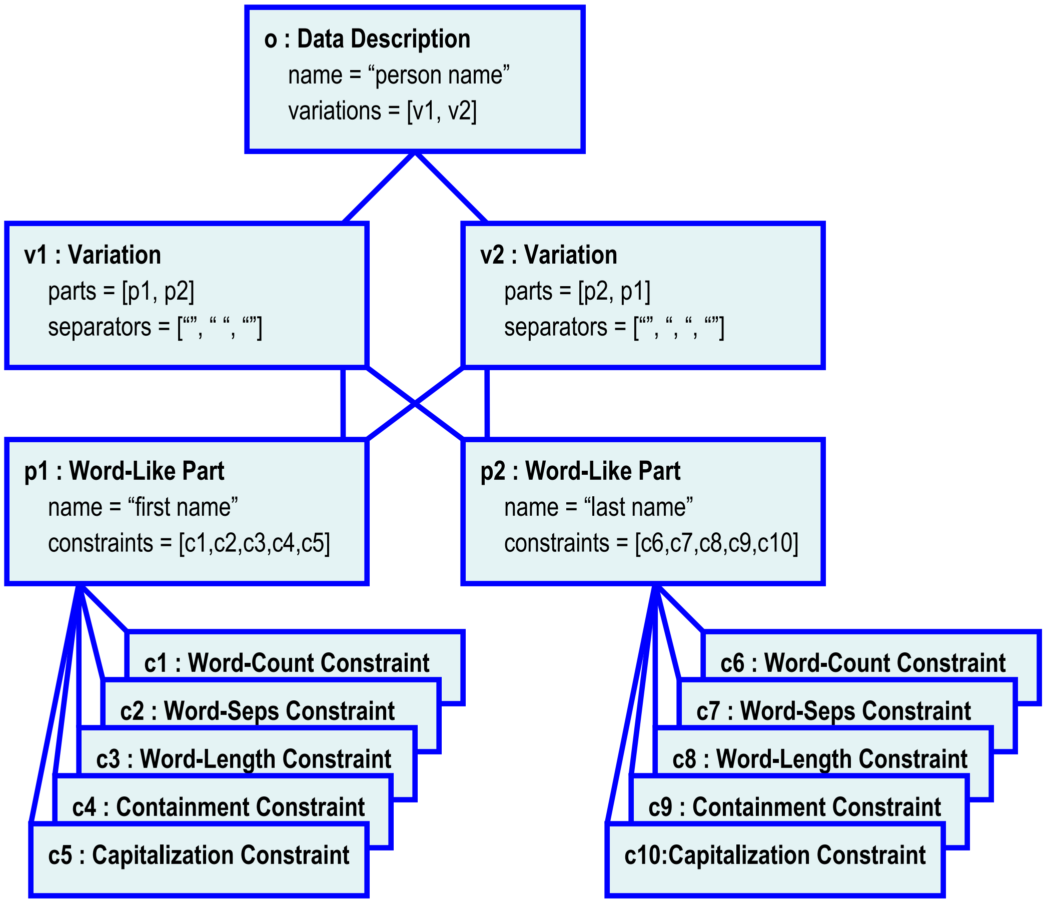


Figure 20. UML object diagram of the data description edited as shown in Figure 18


o : Data Description
name = "person name"
variations = [v1, v2]
v1 : Variation
parts = [p1, p2]
separators = ["", " ", ""]
v2 : Variation
parts = [p2, p1]
separators = ["", ", ", ""]
p1 : Word-Like Part
name = "first name"
constraints = [c1,c2,c3,c4,c5]
p2 : Word-Like Part
name = "last name"
constraints = [c6,c7,c8,c9,c10]
c1 : Word-Count Constraint
c2 : Word-Seps Constraint
c3 : Word-Length Constraint
c4 : Containment Constraint
c5 : Capitalization Constraint
c6 : Word-Count Constraint
c7 : Word-Seps Constraint
c8 : Word-Length Constraint
c9 : Containment Constraint
c10:Capitalization Constraint

Kinds of parts

As noted by Section 3.4, certain kinds of parts commonly occur inside of hierarchical data. These kinds of parts are directly supported by Toped^{++}: Numeric, Word-like, and Hierarchical. Enumeration kinds of data are indirectly supported, as discussed below. Each icon in the Toolbox on the left side of the screen corresponds to a prototype instance of a part or separator. Blue boxes represent parts, while grey boxes represent separators. The user can put separators before and after the list of parts (in addition to between the parts); omitted separators are recorded in the data description as a separator with a value of "". The user can drag and drop separators and part icons to move them around.

When the user drags a new part from the Toolbox and drops it into a variation's row, the part is "pre-loaded" with a default set of constraints that are usually appropriate for that kind of part. The user can click on a part icon to add, edit, or delete part constraints in the box at the lower right corner of the screen. As in users' verbal descriptions of data categories, these constraints might be true most of the time but not always. Specifically, most constraints can be marked as “never”, “rarely”, “often”, “almost always”, or “always” true. These adverbs of frequency were selected 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%, 60%, 90%, and 100%, respectively [76]. This robust correspondence makes it feasible to integrate constraints with formal grammars (Section 5.1).

For Numeric parts, the user can specify that the part is in a certain numeric range, has up to a certain number of decimal digits, has padding with leading zeroes, contains/starts with/ends with certain digits, or matches a whitelist of specific values. For Word-like parts, the user can specify that the part has a certain number of words of a certain length, has certain characters between words, contains/starts with/ends with certain characters, has words with certain capitalization, or matches a whitelist of specific values. For Hierarchical parts, the user can specify that the part matches another data description or matches a whitelist of specific values.

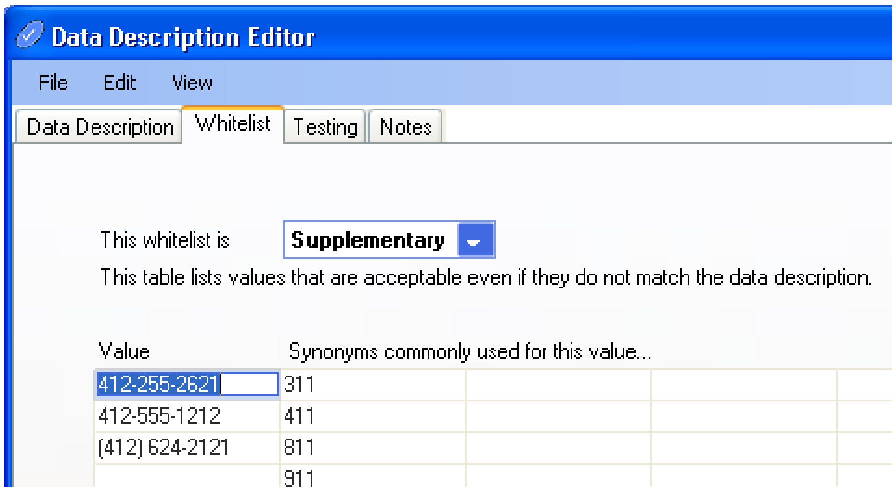
Whitelists

To support cases when parts or the string in its entirety must be in an enumeration, Toped^{++} provides a grid-like user interface for entering a whitelist (Figure 21). The data description as a whole can have a whitelist, and each part individually can have a whitelist, too.

Each whitelist can be used in three different modes. A whitelist in “Supplementary” mode specifies certain strings that should be accepted even if they do not match the main data description. For example, the user could specify that “311”, “411”, “811”, and “911” should be allowed as phone numbers (Figure 21). A whitelist in “Strict” mode specifies that *only* certain strings are allowed (a constraint that must always be satisfied), and any other strings are definitely invalid. Finally, a whitelist in “Approximate” mode specifies that inputs almost always should generally appear in the whitelist, but if they do not happen to appear in the whitelist, then they are merely questionable rather than definitely invalid (a constraint that is often but not always true).

Toped^{++} refers to strings in the second and succeeding columns as “synonyms” for strings in the first column. As described in the next two chapters, a synonym format only accepts the strings that appear in the column, and transformation functions to and from the synonym column are implemented with lookup tables.

Figure 21. Editing a whitelist for phone numbers



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			

Add-ins can offer users the opportunity to not only override error messages for strings, but also to add those strings to the data description's whitelist so that the strings are never marked as errors in the future. As this data description is stored on the user's computer, adding a string to the whitelist will cause the data description to start accepting that string in all applications referencing that data description. Future versions of add-ins in the TDE might track per-application whitelists, which they could use to filter errors identified by the data description. More or less granular whitelists are also possible. For example, the Excel add-in could keep a whitelist of all strings that should be allowed in any spreadsheet for any data description (and would ignore any related error messages coming back from the TDE for those strings), and it could keep a whitelist of strings that should be allowed in any spreadsheet for a particular data description (and would ignore any related error messages coming back from the TDE when validating or reformatting those strings with that data description). Such a profusion of different, partially-overlapping whitelists seemed too complex to include in the TDE prototype of this thesis research, but they might be useful in future versions of the TDE.

Linking constraints

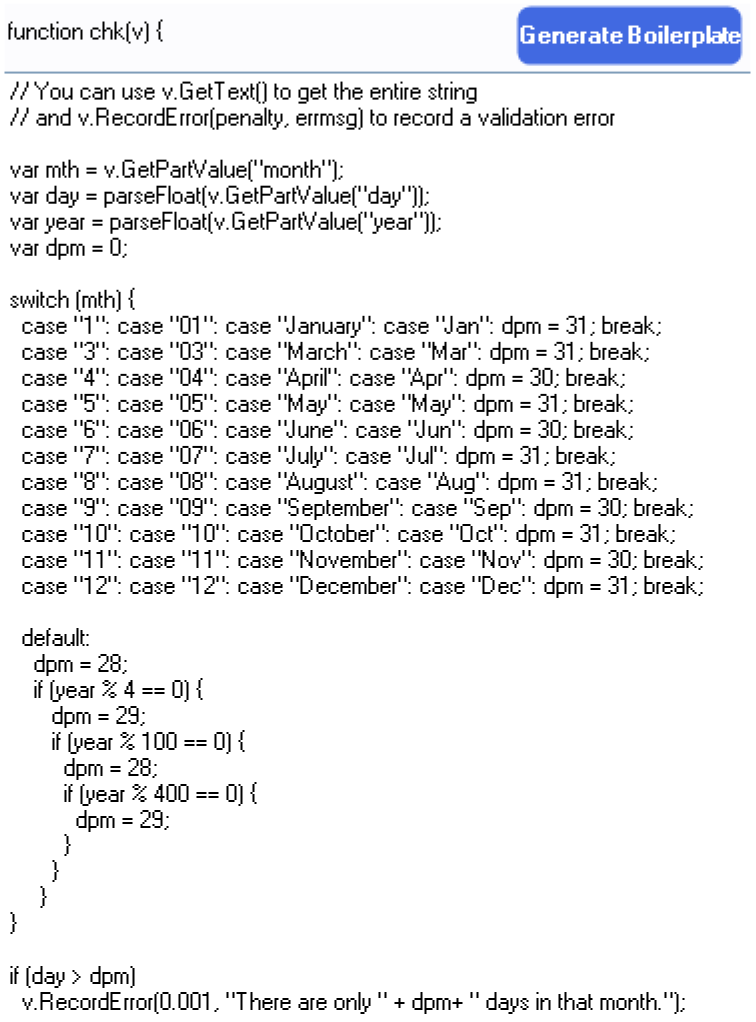
Toped^{++} provides a screen for entering "linking constraints" in JavaScript that check whether the parts are related properly to one another. This is useful for implementing conditional constraints where the valid values of one part depend on the values in another part, such as in dates. Linking constraints are conceptually similar to the constraints supported by the individual parts (as shown in Figure 18), except that the JavaScript-based constraint applies to the data description as a whole rather than to just one part. The user's JavaScript function implementing the linking constraint can internally check as many rules as desired for validating the data. Because the programming language is JavaScript, the full range of Turing completeness is supported.

To edit a data description's linking constraint, the user begins by selecting a menu option under the "View" menu of Toped^{++}. A button is provided to generate boilerplate JavaScript code, in order to help the user see how to programmatically access the values of different parts. Actually coding JavaScript-based constraints is currently very verbose and error-prone, as shown in the date example in Figure 22. (For succinctness, this example only checks whether the day is in a valid range; the month and year are already validated using standard non-JavaScript constraints on those two individual parts, using a whitelist and numeric ranges, respectively.) Fortunately, coding constraints in JavaScript is rarely needed (essentially, only when linking constraints are required, or when the part-specific constraints shown in Figure 19 are inadequate). In particular, JavaScript-based constraints were not needed for any of the kinds of data involved in the user studies (Sections 5.2.3 and 6.2.2).

Testing features

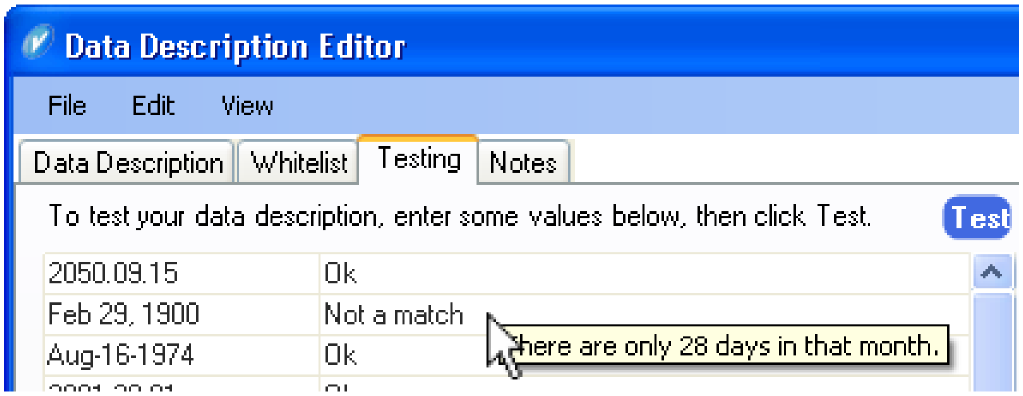
Alongside the tab for editing the data description's whitelist, Toped^{++} provides a "Testing" tab where the user can enter example strings to test against the data description (Figure 23). When the user clicks the "Test" button, the TDE automatically generates a tope implementation from the data description (as described by the next chapter) and uses it to validate each example string. For each invalid or questionable string, Toped^{++} displays a tooltip similar to the ones shown by the Excel add-in. In addition, to help the user test each individual part's constraints, each part icon has a user-editable example string (such a "von Neumann" in the last name part in Figure 18), which is validated using the part's constraints to generate a targeted message in a tooltip if the example fails to meet the constraints.

Figure 22. Editing a linking constraint for validating dates.
The variable v is a helper object. Its `GetPartValue` method returns the string values of a part, and the `RecordError` method records a constraint violation penalty and a human-readable error message. The `parseFloat` method is built into JavaScript. The constraints for the month and year are already checked by non-JavaScript constraints, so this code can assume that those are valid.



```
function chk(v) {

// You can use v.GetText() to get the entire string
// and v.RecordError(penalty, errmsg) to record a validation error

var mth = v.GetPartValue("month");
var day = parseFloat(v.GetPartValue("day"));
var year = parseFloat(v.GetPartValue("year"));
var dpm = 0;

switch (mth) {
  case "1": case "01": case "January": case "Jan": dpm = 31; break;
  case "3": case "03": case "March": case "Mar": dpm = 31; break;
  case "4": case "04": case "April": case "Apr": dpm = 30; break;
  case "5": case "05": case "May": case "May": dpm = 31; break;
  case "6": case "06": case "June": case "Jun": dpm = 30; break;
  case "7": case "07": case "July": case "Jul": dpm = 31; break;
  case "8": case "08": case "August": case "Aug": dpm = 31; break;
  case "9": case "09": case "September": case "Sep": dpm = 30; break;
  case "10": case "10": case "October": case "Oct": dpm = 31; break;
  case "11": case "11": case "November": case "Nov": dpm = 30; break;
  case "12": case "12": case "December": case "Dec": dpm = 31; break;

  default:
    dpm = 28;
    if (year % 4 == 0) {
      dpm = 29;
      if (year % 100 == 0) {
        dpm = 28;
        if (year % 400 == 0) {
          dpm = 29;
        }
      }
    }
}

if (day > dpm)
  v.RecordError(0.001, "There are only " + dpm+ " days in that month.");
```

Figure 23. "Testing" tab in Toped++



To test your data description, enter some values below, then click Test.

2050.09.15	Ok
Feb 29, 1900	Not a match
Aug-16-1974	Ok

There are only 28 days in that month.

The "Testing" tab provides additional, more sophisticated features for semi-automatically identifying potential problems with data descriptions (not shown in Figure 23, as they are farther down the screen). Since users are probably most likely to encounter these problems when attempting to reuse existing data descriptions, particularly when combining one or more unfamiliar data descriptions created by other users, these features are discussed within the context of reuse by Section 7.1.3.

4.4 Inferring boilerplate data descriptions

While the user could create a data description from scratch, in practice add-ins gently encourage the user to provide examples of the strings to validate, which the add-ins pass to the Topei module. This module infers a preliminary data description that describes the examples as best as the inference algorithm can do automatically. The TDE passes this data description to the editor for review and customization by the user.

Topei infers a data description in two phases. First, it identifies the data's parts. Second, it identifies constraints on each part. The discussion below will demonstrate inference on the following example university courses:

BIO-101
Math 340
CS-380
cs-202
LIT 373
Introduction to American History
SPA-203

Identifying parts in variations

In the first phase of inference, Topei replaces each alphanumeric character with its character class—uppercase, lowercase, or digit (shown as "A", "a", or "0" below).

```
AAA-000

Aaaa 000

AA-000

aa-000

AAA 000

Aaaaaaaaaaaa aa Aaaaaaaa Aaaaaaa

AAA-000
```

Next, Topei collapses runs of characters into number and word tokens. It generates a numeric part (shown as “N” below) for each number token and combines multiple word tokens into a word-like part (shown as “W” below) if they are delimited by certain punctuation (“ ”, “-”, “&”, “,”, “.”); such punctuation characters often appear between words in multi-word parts such as names of companies.

```
W-N

W N

W-N

W-N

W N

W

W-N
```

Finally, Topei creates a variation for each distinct sequence of parts and separators and discards any variations that have a different sequence of parts than the most common variation. Thus, the resulting variations all have the same parts (in the same order) but different separators. In the running example, Topei would retain two variations and discard the `W` variation.

```
W-N

W N
```

Identifying constraints on parts

Having matched substrings of the examples to parts, Topei next uses those substrings to identify constraints.

For numeric parts, Topei creates a mandatory numeric constraint that allows the full range of values demonstrated by the examples. These numeric boundaries are not tight, either: because so many kinds of data (such as the course numbers shown above) have numeric parts that have a certain number of digits, Topei creates a numeric constraint that allows any number of digits demonstrated by the examples. For example, based on the strings above, Topei would constrain the course number's second part to the range 100-999, since all of the examples have 3 digits. If one of the examples happened to have 5 digits, then the would constrain the course number's second part to the range 100-99999.

For Word-like parts, Topei creates a mandatory constraint that allows the full range of the number of characters in each word. However, if at least 80% of the examples demonstrate a specific length, then the constraint instead requires that specific length. Finally, Topei creates a mandatory constraint on the number of words based on the range of counts demonstrated by the examples. In the course numbers above, the examples of department in the `W-N` and `W N` variations were "BIO", "Math", "CS", "cs", "LIT", and "SPA". Thus, the inferred word-like part requires 1 word. Fewer than 80% of the example words have a specific length—the most common length, 3, only accounts for 50% of the examples—so Topei infers a length constraint that allows the full range of lengths, which is 2-4 letters per word.

The constraints inferred by this latest version of Topei are fairly loose in the sense that they cover all of the examples of the variations (except in the case above when there is a preponderance of evidence that the number of letters in a word should match a fixed length). In contrast, an earlier version of Topei inferred tighter bounds that covered 95% of the examples rather than all of the examples. The intent behind this was to facilitate outlier finding, under the expectation that a small fraction of the user-provided examples would be invalid. However, after I personally used that version of Topei for over a year, I decided to loosen the inferred constraints, as described above. There were two usability-related reasons for this change.

The first reason for broadening the inferred constraints is that although tighter bounds generally flagged more strings as invalid, giving the resulting validation a better (lower) false negative rate, it also caused a worse (higher) false positive rate. Unfortunately, the worsening of false positives often seemed to be disproportionate to the improvement in false negatives. In other words, that last 5% of outliers often turn out upon inspection to be valid, meaning that real data often contains a lot of unusual but valid values. Correcting an over-zealous constraint requires the user to iteratively broaden the constraints until it stops issuing false positives. This requires extra effort. If anything, future versions of Topei might *broaden* the constraints even further beyond those demonstrated by the user-provided examples, under the assumption that the user-provided examples do not adequately demonstrate the full range of valid values that the resulting tope implementation might check.

The second reason for broadening the inferred constraints to cover all of the examples is simply that when Topei infers something slightly different from the examples, this provokes a feeling of "where did *that* come from?" In terms of human-computer interaction, this mismatch between the examples and the inferred patterns widens the Gulf of Execution—making it more difficult for the user to predict what inputs should be provided in order to attain a certain computer state [81].

For the most part, similar conclusions about the desirability of broad constraints are reflected in the design of the constraint-inference systems discussed in the Related Work (Section 2.7). For example, when inferring patterns from example strings, Lapis only infers constraints that highlight extremely few outliers, as the system's designers believed that "highlighting a large number of outliers is unhelpful to the user, since the user must examine each one" [70]. It would be fine to draw a large number of outliers to the user's attention if those outliers generally were invalid, but the designers' empirical studies show that the number of invalid outliers is usually relatively low, and experiments with the TDE show similar results (Section 5.2.2).

Of all the inference algorithms in systems mentioned by the Related Work, the one that most different from the TDE's is that of DataProG [59]. This regular expression inference algorithm tunes constraints by tightening them until they cease to satisfy statistical tests. For example, suppose that the example strings were "peel", "apple", and "leap", which together only demonstrate five different lowercase letters. Is this enough evidence to justify inferring a regular expression that only allows these five different letters? Or should DataProG instead produce a regular expression that allows any lowercase letter? To make this decision, DataProG applies a statistical test, and the result would depend on how many examples were provided and their distribution. The null hypothesis for these tests is based on the assumption that all strings are independently sampled from a uniform probability distribution. In this example, the null hypothesis is that all 14 letters in the three examples were randomly chosen from the 26 different lowercase letters of the alphabet. In this case, it is extremely unlikely—around 1 in a million—that these 14 letters would only contain five distinct values, so DataProG would reject the null hypothese and select the tighter constraint.

One serious practical problem with using this approach in the TDE is that if a value appears in one spreadsheet cell or database row, then that value also tends to appear in nearby cells or rows. Thus, real data values encountered or provided by end users have a non-zero correlation, which violates DataProG's most basic independence assumption. Therefore, assuming a uniform probability distribution could lead to inferring too many constraints and excessively tight constraints. This fact and the two usability-related reasons explained above argue in favor of the approach currently implemented in the latest version of Topei.

4.5 Correspondence between data descriptions' variations and topes' formats

The next two chapters will describe how the TDE's Topeg module generates `isa` and `trf` function implementations. A precondition to understanding these implementations is understanding how Topeg maps from data descriptions to formats.

The general relationship between a data description and the automatically generated tope implementation is that each variation corresponds to one or more of the tope implementation's 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 capitalization variations, depending on the user's specification (as indicated by the user by selecting checkboxes as shown in Figure 18), and each Numeric part has one variation for each number of decimal digits specified as allowable by the user. For example, in Figure 18, the user 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 could match "John von Neumann", "JOHN von Neumann", "John VON NEUMANN", and "JOHN VON NEUMANN". The second variation corresponds to four formats, as well (with the two names in reversed order).

A Hierarchical part can also have variations. Since a Hierarchical part references another data description, the part's variations are those of the referenced data description. For example, a user might have a spreadsheet cell containing a person's online 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 eight 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", "bam@CS.CMU.EDU", and "BAM@CS.CMU.EDU"). If the email address happened to have three formats, then the overall data description would have 8 x 3 = 24 formats.

In general, each format 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.

Because a part in Toped^{++} can reference multiple formats, it is possible to quickly build up quite complex validation code. For example, a month data description might have three one-part variations (one to recognize "August", another 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 six 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 encountered on a regular basis, 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.

Chapter 5. Validation functions

Chapter 4 described the editing tools provided in the Tope Development Environment (TDE) so that users may create data descriptions. Each data description contains a list of variations, each of which has a list of constrained parts with intervening separators. The TDE does not validate strings directly with a data description. Instead, the TDE's Topeg module generates a tope implementation from the data description, then calls the tope implementation's `isa` functions to validate strings.

Thus, there are two important topics related to validation. One is the issue of how the TDE generates `isa` functions, a process which is preparatory to actually validating strings. The second topic is what steps the `isa` functions actually take at runtime to validate strings. As discussed by Section 5.1, the general idea is that the TDE creates `isa` functions that each include an automatically-generated grammar; at runtime, each `isa` function tests input strings against its grammar.

The initial motivation for this research was to help users to automate validating and reformatting operations (as described by Chapter 2). These operations often involved strings located in web forms and spreadsheets. Thus, in the context of this chapter, the primary evaluation criterion should be to test whether or not users can actually use the TDE to validate strings from web forms and spreadsheets.

Section 5.2 describes this evaluation in two parts. First, it explores how well an expert user (me) can use the TDE to validate strings from web forms and spreadsheets. The evaluation shows that an expert user can express validation that is three times as accurate as the commonly-practiced validation method based on using commonly-available regular expressions. Most of this benefit comes from the simplicity of packing many formats into a data description, with a smaller benefit coming from the fact that questionable inputs can be double-checked and only rejected if they are truly invalid.

Second, Section 5.2 examines how well an ordinary end user can use the TDE to validate strings from spreadsheets, compared to using an alternate system, Lapis [70]. The evaluation shows that with the TDE, users can validate spreadsheet data twice as fast and find three times as many invalid strings as they can with the existing Lapis system.

Qualitative analysis suggests that much of this benefit comes from the fact that the TDE provides many programmatic primitives that directly correspond to frequently-occurring patterns in data categories. In human-computer interaction terms, primitives like these give the TDE high directness, tight closeness of mapping, and a small Gulf of Execution [37][81], simplifying the task of translating a mental description of data categories into the user interface's visual notation, and vice versa.

5.1 TDE support for implementing and executing validation functions

Add-ins validate strings by passing them into `isa` functions. Topeg implements these functions by generating a context-free grammar for each of a tope's formats, then attaching the constraints from the data description onto the grammar's production rules, resulting in an "augmented" context-free grammar. Topeg embeds the grammar for each format within the respective `isa` function. At runtime, the function validates strings by parsing them according to its grammar, returning a value between 0 and 1 depending on whether the parse succeeds and whether the string violates production constraints. Context-free grammars were chosen as a basis for this implementation because existing context-free parsing algorithms were readily extendible to support soft production constraints.

5.1.1 Generating an augmented context-free grammar

To create a format's grammar, Topeg generates a hierarchy of context-free grammar productions to represent Hierarchical parts, indicating that a part should match another tope. It inserts separators between parts then generates leaf productions for parts based on each part's Number or Word-like constraints. For example, if a Word-like part always contains 1-6 words with 1-8 lowercase letters per word and only hyphens or ampersands between words, then Topeg would *notionally* generate a grammar like the following (with `#` prefixing variables):

```
#PART : #WORD

#PART : #WORD #SEP #WORD
```

```
#PART : #WORD #SEP #WORD #SEP #WORD

#PART : #WORD #SEP #WORD #SEP #WORD #SEP #WORD

#PART : #WORD #SEP #WORD #SEP #WORD #SEP #WORD #SEP #WORD

#PART : #WORD #SEP #WORD #SEP #WORD #SEP #WORD #SEP #WORD #SEP #WORD

#WORD : #CH

#WORD : #CH #CH

#WORD : #CH #CH #CH

#WORD : #CH #CH #CH #CH

#WORD : #CH #CH #CH #CH #CH

#WORD : #CH #CH #CH #CH #CH #CH

#WORD : #CH #CH #CH #CH #CH #CH #CH

#WORD : #CH #CH #CH #CH #CH #CH #CH #CH

#CH : a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

#SEP : - | &
```

The example grammar above is only notional because for compactness, Topeg actually enforces most constraints by augmenting productions with constraints, rather than incorporating every constraint into productions. When parsing strings, these constraints can be checked as the productions fire, thereby identifying parses that violate constraints (as described by Section 5.1.2). For example, the generated #WORD production would allow any number of characters, but Topeg would attach a constraint to the production to verify at runtime that the word has an acceptable number of letters. Linking constraints (written in JavaScript, as described by Section 4.3) would be attached to the data description's top-level production, rather than the productions for individual parts.

This approach has four benefits.

The first benefit of augmenting productions with constraints is that the resulting grammars generally have fewer productions than they would if all constraints were directly incorporated the productions. For example, all of the productions for #WORD in the grammar above could be replaced with a single production. Specifically, if a Word-like part almost always contains 1-6 words that each always have 1-8 lowercase letters per word and only hyphens or ampersands between words, then Topeg would generate a grammar like the following:

```
#PART : #WORDLIST : COUNT(#WORD)>=1 && COUNT(#WORD)<=6 {90}
```

```
#WORDLIST : #WORD | #WORD #SEP #WORDLIST

#WORD : #CHLIST : COUNT(#CH)>=1 && COUNT(#CH)<=8 {100}

#CHLIST : #CH | #CH #CHLIST

#CH : a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

#SEP : - | &
```

The constraint in the first line above is pseudo-code indicating that the number of #WORD nodes in the parse subtree rooted at the #PART node should almost always be between 1 and 6, inclusive. The third line indicates that the number of #CH nodes in the subtree rooted at the #WORD node should always be between 1 and 8, inclusive. The grammar above shows pseudo-code for the constraints because they are actually implemented in much more verbose C# or Java (as the TDE is available in either language). In the case of these two constraints, this code implements an attribute grammar [49] that computes a bottom-up count of the number of #WORD nodes in the subtree. (In the case of the C# implementation of the TDE, constraints written in JavaScript are simply wrapped with C# that calls the Microsoft JavaScript interpreter via a COM object provided by Microsoft; since no such support for JavaScript was readily available for Java, the Java-based implementation of the TDE simply throws an exception if it encounters a linking constraint coded in JavaScript.)

The second benefit of augmenting productions with constraints implemented in C# or Java code is that the TDE supports the full range of Turing completeness internally. Of course, unless the user wants to resort to writing JavaScript in a linking constraint, it is only possible to express the non-JavaScript constraints exposed by the Toped or Toped^{++} user interface. However, the ability to put any computable constraint into the grammar makes it feasible to support new kinds of parts besides Numeric and Word-like, as well as new kinds of constraints, should they become desirable in the future.

The third benefit of augmenting productions with constraints is that it provides a straightforward mechanism for identifying questionable values based on soft constraints. For example, perhaps the Word-like part "almost always" has 1-6 words. Therefore, it could theoretically contain more than 6 words, and the production should allow an unlimited number of words. The production would then be augmented with a constraint that checked whether the actual number of words in a parse actually fall into the specified range. If this constraint is violated, then the parse can be flagged as questionable. In particular, Topeg annotates each soft constraint with a number corresponding to the probability that people generally associate with the constraint's adverb of frequency (10% for "rarely", 60% for "often", and 90% for "almost always") [76]. Thus, as described by Section 5.1, inputs can be flagged as "more questionable" if they violate stronger constraints.

The fourth and final benefit of this approach is that it can compute "null" constraints (that are always satisfied) whose sole purpose is to track the parse as it proceeds. In particular, these null constraints can track which separators are used in a parse tree and what permutation of the parts actually appeared in the input string. This is precisely the information required to determine what format corresponds to the parse tree, since each format is uniquely distinguished by the presence and ordering of parts and separators (Section 4.5).

This last benefit allows Topeg to coalesce the individual format grammars into a single "subsuming" grammar for the tope *as a whole*. When validating against the entire tope (rather than a particular format), a string can be tested against this subsuming grammar. In effect, this computes the overall tope validation function $\phi(s) = \max_{f \in F}[\mathtt{isa}_f(s)]$. Thus, although add-ins could iterate through every format in a tope, calling each `isa` in succession to find the maximal `isa` score, they alternatively can call ϕ directly. Calling ϕ simplifies the code required for add-ins (compared to iterating over all `isa` functions) and generally improves performance.[3]

The discussion above has laid out four benefits of generating an augmented context-free grammar rather than incorporating every constraint directly into the context-free grammar. The primary disadvantage of this approach is that although the process for generating an augmented context-free grammar is intellectually straightforward, the actual code required inside of Topeg is extremely detailed and several thousand lines long, which made it difficult to code, test, and debug.

[3] The subsuming grammar is also useful for determining the "source format" that should be used for transforming a string (described further in Chapter 6).

5.1.2 Parsing strings with an augmented context-free grammar

At runtime, each `isa` function parses strings according its augmented grammar. Since all functions use the same parsing algorithm, the actual parsing code (which takes the grammar and the string as parameters) is factored out into a parsing library called "Topep". This library is based on GLR [120], 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 by Topeg.

Unlike GLR, Topep supports constraints on productions. When a completed production produces a variable `v`, GLR always treats `v` as valid and uses it to complete other productions waiting for `v`. In contrast, Topep associates a score with each variable instance (a parse tree node), ranging from 0 through 1. When a production `p` for `v` is completed, the parser evaluates constraints on `p`. If `v` violates a constraint, then Topep downgrades the score of `v` by multiplying it by the (estimated) probability that the constraint would be violated by chance, based on the adverb of frequency selected by the user for how often the constraint should be true. For example, violating a constraint that is often true (i.e.: 60% of the time) would result in multiplying the score by 0.4. Violating a constraint that is almost always true (i.e.: 90% of the time) would result in multiplying the score by 0.1. Since Topep downgrades scores by multiplication, violations of two often-true constraints would result in a score of 0.4 * 0.4 = 0.16.

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, consistent with the tope model.

Multiplying the scores is appropriate because it sets a conservative bound on the parent's score. Formally, if two nodes had independent probabilities x_1 and x_2 of violating their constraints, then the correct probability of both of them violating their constraints would be the product $x_1 \cdot x_2$ But perhaps these constraints' violations are correlated, so x_1 and x_2 are not independent. In that case, the product of probabilities provides a lower bound on the probability of both being violated. Thus, the resulting `isa` score might be closer to 0 (more pessimistic) than the input justifies.

Reporting this conservative number is appropriate because the worst that happens is that a value is flagged as more questionable than it actually is. As a result, users might be asked to double-check values that do not really need attention. However, in practice this does not turn out to be a problem because at worst, it can only cause a *questionable* input to appear *more questionable* than it really is. As discussed by Section 5.2.2, only a few percent of strings are questionable at all. Thus, empirically, this conservatism of multiplying probabilities will cause users to double-check at worst only a few more percent of the strings than they otherwise would if Topep had used a less conservative estimate.[4]

Of course, this approach of multiplying probabilities could be refined in future versions of the TDE, as discussed by Chapter 8. In particular, the TDE could track how often different constraints are simultaneously violated, yielding a joint probability of violation. This measured probability could then be used in place of the product of individual constraint probabilities specified by the user, making the resulting `isa` function precisely equal to the probability that the provided string is valid.

As the parse proceeds, Topep tracks a list of violated constraints and concatenates their corresponding English equivalents into an error message, as shown in Figure 12 and Figure 23. This English equivalent is essentially the same as the text presented by Toped^{++} to users. When a parse totally fails, making it impossible to identify specific violated constraints, Topep generates a message by examining the waiting (unfulfilled) grammar productions and concatenating the constraints named in Toped^{++} that were used to generate those productions. The resulting message is much more targeted and descriptive than 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.

[4] The statistical independence assumption embodied in Topep is similar to the one rejected while designing Topei (Section 4.4). The reason for accepting the assumption in one case but rejecting it in the other is that the assumption has a much smaller potential scope of effect in the context of Topep than it does in the context of Topei. In Topep, at worst, the assumption could only affect the few inputs already known to be questionable. In Topei, in contrast, the assumption could affect every valid string, as the assumption could result in inferring too many constraints.

5.1.3 Implementing validation rules with the early Toped prototype

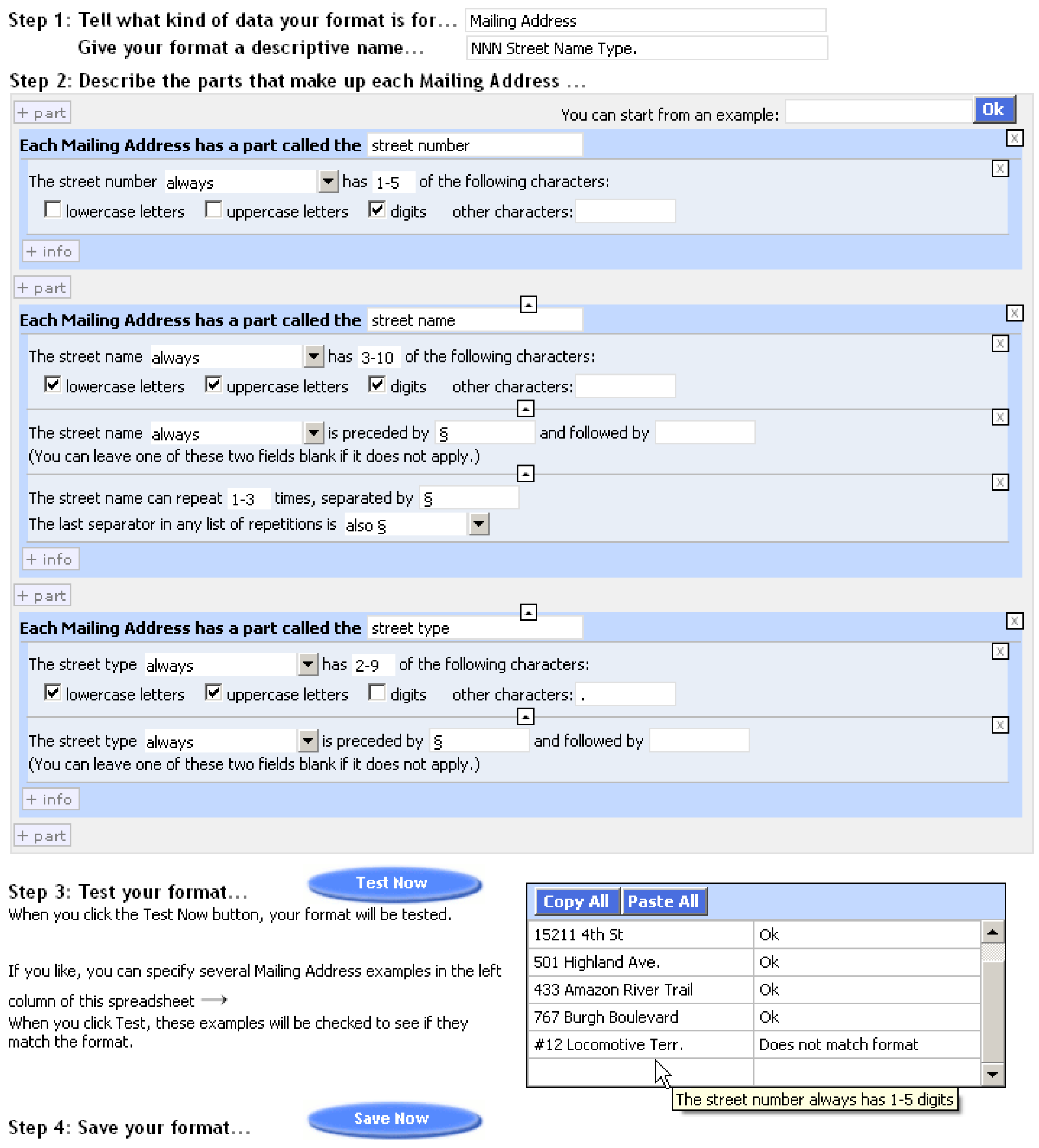
Toped^{++} is actually the second major version of the TDE's data description editor. The initial version, called Toped, was a first attempt at enabling a user to describe the constrained parts in a kind of data, as well as their sequence in different formats. The biggest difference between Toped and Toped^{++} is that Toped required users to explicitly edit a description of each format and each inter-format transformation. (The transformation editor is described in detail by Section 6.1.3.) For example, if a tope has three formats with six intervening transformations, then the user would need to edit a description for the first format, then edit a description for the second format, then a description for the third… and so on. Thus, creating a multi-format tope with Toped can be extremely tedious.

This did not actually create any usability problems in the lab experiment described by Section 5.2.3 below, since users could complete the tasks in that experiment by implementing a single-format tope for each data category. Nonetheless, users identified a few problems with Toped, which I addressed in Toped^{++}. Moreover, while I was creating the multi-format data descriptions needed for the evaluations discussed by Sections 5.2.1 and 5.2.2, the tediousness of manually creating many formats made it painfully obvious that the TDE needed a more user-friendly way of implementing multi-format topes. The main reason was that when the user added constraints to one format, he then had to manually add the same constraints to the other formats. That is, each part did not "carry along" its constraints from format to format. These considerations motivated the development of Toped^{++}, which displays the entire data description at once on the screen. Toped^{++} strongly associates constraints with their respective parts so that each part has the same constraints in all the formats where it appears. This new tool was ready in time for the usability study described by Section 6.2.2. That study included reformatting and validation activities, ultimately confirming that the new tool was effective at helping users with their work.

Details of the Toped editor

The format editor of Toped is a form-based user interface with four sections (Figure 24).

Figure 24. Editing a format description in Toped.
Adding, removing, and reordering parts/constraints is accomplished with the +, ☒, and ⏶ buttons. Each § indicates a space.



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 ...

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:

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 §

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.)

Step 3: Test your format...
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.

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...

First, the user gives the data description a name such as “phone number” or “person name” and names the format. If the user is adding a new format to an existing data description, then Toped initializes the textbox for the data description name with the existing value (and propagates any edits back to the data description).

Second, the user defines the format’s parts, each of which is presented as a section in the form. For example, a US phone number has three parts: area code, exchange, and local number. The user can add, remove and reorder parts, and he can specify constraints on parts.

In the third step of the editor, the user 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 user can iteratively debug the format description.

Finally, Toped has a button so the user can save the data description.

Constraints

Table 9 shows the available constraints, which were initially identified based on a review of the data commonly encountered by users (Sections 2.3 and 2.4) and based on asking administrative assistants to describe data (Section 2.5). Toped does not distinguish between different kinds of parts, so users can mix and match constraints at will (even in non-sensical combinations).

Users may mark the Pattern, Literal, Numeric, Wrapped, and Substring constraints as “never”, “rarely”, “often”, “almost always”, or “always” true. These adverbs of frequency were selected 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%, 60%, 90%, and 100%, respectively [76]. This robust correspondence makes it feasible to integrate constraints with formal grammars (Section 5.1). Toped has an advanced mode, not shown in Figure 24, for specifying linking constraints that span multiple parts (Section 3.4), such as the intricate leap year rules for dates.

Table 9. Constraints that can be applied to parts in Toped.

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

Earlier studies suggested that end users sometimes have trouble grasping mixtures of conjunction and disjunction [77]. Consequently, all constraints are conjoined in Toped, with the exception that disjunction is implied when two constraints have parallel sentence structure but are each not always true. For example, a user could specify that the area code in a phone number is *often* followed by a "-", and that the area code is *often* followed by a ") ". During the user study with Toped (Section 5.2.3), a single participant commented that this design choice was unintuitive.

Upon further consideration of the situations in which disjunction was necessary, it became clear that this kind of disjunction resulted from attempts to conflate two different formats into one format description (e.g.: formats for recognizing "800-777-8080" and also "(800) 777-8080"). Essentially, this conflates the *separators between parts* with the *constraints on parts*. In Toped++, this conflation is removed by providing icons for specifying separators and eliminating the "Wrapped" constraint. Since moving to the new interface (and using Toped^{++} in my daily work for over a year), I have never had a need for a disjunction of two constraints, even when I used Toped^{++} to implement topes for the 32 most common data categories (Section 7.2.2). This has confirmed the appropriateness of using a pure conjunction of constraints in Toped^{++}.

Moving parts around

Users can move a part specification in Toped by clicking a special ⏶ button. This slides the part specification upward on the screen (vertically). There are two problems with this layout and interaction. First, laying out the part specification vertically on the screen does not match the horizontal orientation of parts as they naturally appear in real strings; some other researchers (though no study participants) have commented that this is unintuitive. Second, moving a part in Toped more than one “slot” requires multiple clicks (with intervening mouse moves), which is fairly slow because it requires the user to aim the mouse several times, leading to high viscosity [37]. Toped^{++} improves the situation by laying out part icons horizontally and allowing the user to drag and drop icons anywhere on the screen.

Making spaces visible

One problem with some editing tools is that spaces are invisible and therefore could be hard to debug. To counter this, Toped makes spaces visible by representing them with a special symbol, §. Though using a special space marker slightly reduces readability, it is preferable to having spaces that the user cannot see or debug. (In error messages, spaces appear as SPACE to avoid font-related problems.)

In response to an early (rejected) paper that described Toped, one reviewer commented that this symbol commonly is used as a “section mark” and recommended using a very wide underscore instead. This suggestion is incorporated into Toped^{++}, but by default, Toped++ does not show space markers since most spaces appear in inter-part separators, which are plainly visible without a special marker because they are shown as grey icons. Users can activate space markers by selecting an option under the “View” menu of Toped^{++}.

Integration with Topei

To help the end user get started with creating a format description, Toped includes a textbox that accepts an example string. There is also a popup window where the user can enter more than one example. Toped calls an early prototype of Topei with these examples to produce a boilerplate format description.

Since Toped can only display one format description at a time, it discards all formats identified by Topei except for the most common format in the most common variation. In addition, since Toped does not distinguish between Word-like and Numeric parts, the early prototype of Topei treats each W and N as a generic part that could contain letters or numbers, depending on what characters were present in the examples. In addition, the early version of Topei lacks code for combining different words into a single part.

For example, if a user provided the strings on the left below, the early Topei prototype would infer the formats shown on the right (where `x` is a generic part) …

BIO-101	`x-x`
Math 340	`x x`
CS-380	`x-x`
cs-202	`x-x`
LIT 373	`x x`
Introduction to American History	`x x x x`
SPA-203	`x-x`

Of these inferred formats, Toped would retain the most common (`x-x`) and present it for editing. Because this early prototype discarded all but one format, the user had to initialize each format one at a time. The example strings above would suffice for initializing a format that matched "BIO-101", but then to initialize a format that matched "BIO 101", the user would need to provide a new set of examples.

Toped^{++} offers improved integration with a newer version of Topei. The new system presents an entire data description for editing at once rather than one format at a time, it distinguishes between different kinds of parts, and it can identify runs of words as a single word-like part. Thus, as detailed by Section 4.4, the TDE would now extract two variations (`W-N` and `W N`) from the examples above.

5.2 Evaluation

There are two key questions related to evaluating the TDE's support for validating strings. First, is it even possible at all for an expert to express useful validation operations in the TDE? Second, if the TDE were put into the hands of ordinary end users, would it help them to more quickly and effectively validate strings than they can with current practices? The first question focuses purely on the expressiveness of the tool (whether it is possible to use the TDE to accurately validate strings using the algorithms described in Section 5.1), a sort of "best case" scenario that ignores usability. The second question adds human users to the mix (taking into account the user interface described in Chapter 4), which is essential for getting a realistic picture of the TDE's benefit in practice.

In answering the first question regarding whether an expert can "express useful validation", the emphasis is on the word *useful* rather than *express*. The reason is that expressiveness has no value within this context unless it ultimately leads toward the goal of helping end users to validate inputs. In this light, I evaluated expressiveness by using the TDE to implement validation operations for data obtained from real web forms and spreadsheets, then measuring how well those operations accurately differentiated between valid and invalid inputs. These experiments show that it is possible to express topes that are quite accurate at validating strings (Sections 5.2.1 and 5.2.2). In particular, the experiment with spreadsheet data shows that the tope-based validation is much more accurate than current validation approaches based on regular expressions (Section 5.2.2). Most benefit comes from the fact that a tope can recognize and validate multiple formats of input, while a smaller benefit comes from support for identifying questionable values.

The second stage of this evaluation addresses whether putting the TDE in the hands of end users actually enables them to quickly and effectively validate strings. A between-subjects laboratory experiment answers this question by having some users validate data with the TDE and having comparable users validate the same data with the best available alternate system, Lapis [70]. The TDE users completed their work twice as fast as the control subjects, and they were able to find three times as many invalid inputs. TDE users were also more confident in their ability to complete these tasks and reported substantially higher user satisfaction compared to the control subjects. These results strongly suggest that the TDE enables users to validate strings much more quickly and effectively than is possible with current approaches.

The evaluations described by this section used the Toped prototype of the data description editor. Since Toped^{++} provides many usability-oriented features omitted by Toped (such as drag-and-drop editing, as well as support for carrying constraints from part to part), the TDE's final Toped^{++} tool is arguably even more usable than Toped. Thus, the evaluation in this section effectively provides a lower bound on the benefits that the TDE would provide in practice for string validation. Moreover, Toped^{++} was ready for the user experiment involving reformatting (discussed by Section 6.2.2), which confirmed the usability of Toped^{++} for string reformatting.

5.2.1 Expressiveness for validating web form data

To obtain test data for evaluating the expressiveness of the TDE, logging software recorded four administrative assistant's inputs to web forms for three weeks. 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 regular expression describing the string that the user entered. (It recorded a regular expression rather than the literal string in order to protect users' privacy.) To generate regular expressions, it converted each lowercase letter to the regular expression [a-z], uppercase to [A-Z], and digit to [0-9], then concatenated regular expressions and coalesced repeated regular expressions (e.g.: "user0@XYZ.EDU" → "[a-z]{4}[0-9]@[A-Z]{3}.[A-Z]{3}").

The 5897 logged regular expressions were manually examined and then grouped by scripts into semantic data categories such as "email" and "currency" based on HTML names and human-readable text near the fields. As shown in Table 10, 5527 (93.7%) fell into one of 19 categories.

Using the regular expressions as a reference, I created data descriptions in Toped for the 14 asterisked categories, omitting three (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." I omitted two other groups (usernames and passwords) so that I could post formats online without revealing formats of our users' authentication credentials. Finally, I tested the formats with sample strings generated by referring to concrete regular expressions in the log and by using my personal knowledge of formats' semantics (such as what might constitute an email address).

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

Table 10. Data categories gathered from user data. Asterisked groups were used for testing.

Category	Strings	Example regular expression 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	

Testing formats revealed that I committed four user errors. Three were cases of failing to mark a part as optional. The fourth error was an apparent slip of the mouse indicating that a constraint was often true rather than always true. The version of Toped used for this evaluation did not yet have the testing feature. This new testing feature was largely motivated by the recognition that these four errors probably could have been found by testing formats along the way.

After correcting these errors, the 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 four cases that probably were not typos by the users (that is, they were intentionally typed), yet might have been invalid inputs, nonetheless. For example, in two 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 Toped offered no way to express that a street type may contain a period but only in the last position, a limitation addressed in

Toped^{++}. Formats' effectiveness at identifying invalid values suggests that they are powerful enough for validating a variety of data commonly found in web forms.

5.2.2 Expressiveness for validating spreadsheet data

To assess the expressiveness of the TDE for validating spreadsheet data, I created data descriptions in Toped based on the 720 spreadsheets in the EUSES Spreadsheet Corpus's "database" section, which contains a high concentration of string data [33]. As with the web form data above, I analyzed the spreadsheet data and classified it into categories, and then I implemented topes to validate the 32 most common categories.

Identifying data categories

Each spreadsheet column in the EUSES corpus typically contains values from one category, so columns were the unit of analysis for identifying data categories. To focus the evaluation on string data, columns were only used if they contained at least 20 string cells (i.e.: cells that Excel did not represent as a date or number), yielding 4250 columns. Hierarchical agglomerative clustering provided a preliminary grouping into categories [26]. This algorithm used a between-column similarity measure based on a weighted combination of five features: exact match of the column's first cell (which is typically a label), case-insensitive match of the first cell, words in common within the first cell (after word stemming [88]), values in common within cells other than the first cell, and "signatures" in common. The signatures algorithm replaced lowercase letters with 'a', uppercase with 'A', and digits with '0', then collapsed runs of duplicate characters (as in the first phase of Topei, described by Section 4.4).

The algorithm yielded 562 automatically-generated clusters containing at least 2 columns each, for a total of 2598 columns. After I visually inspected these clusters and examined spreadsheets as needed to understand each column's data, it became clear that some clusters could be further merged into larger clusters. For example, the columns with headers "cognome", "faculty", and "author" all contained person last names.

Of the 2598 columns covered by automatically-generated clusters, 531 (20%) did not refer to named entities nor seemed to obey any implicit constraints; this was generic "text" and was discarded as in the web form study (Section 5.2.1). In addition, 196 columns (8%) contained database data dictionary information, such as variable names and variable types. These data appeared to be automatic extracts from database systems, rather than user-entered data, and seemed too clean to be a reasonable test of topes, so these columns were discarded. In addition, 76 columns (3%) contained text labels—that is, literal strings used to help people interpret the "real" data in the spreadsheet—so these were discarded. Finally, 82 columns (3%) were discarded because their meaning was unclear. A total 1713 columns remained, grouped into 246 clusters that each corresponded to a data category.

The most common 32 categories covered 1199 columns. These included Booleans (such as "yes" or "x" to indicate true, and "no" or blank to indicate false), countries, organizations, and person names. The tail outside of these 32 categories included the National Stock Number (a hierarchical pattern used by the US government for requisitions), military ranks for the US Marines, and the standard resource identifier codes stipulated by the government's Energy Analysis and Diagnostic Centers. Each of these tail columns is domain-specific and only occurs in a few columns.

Despite the relative infrequency at which each of these tail columns occurs, together they cover 3051 of the 4250 columns originally extracted from the spreadsheet corpus. There simply are a huge number of domain-specific kinds of data. This fact empirically supports a comment made by Section 1.3, that application developers will be unable to create abstractions for every user's domain-specific category. The fact also highlights the crucial importance of providing a system that people can use to create their own domain-specific tope implementations.

While the 32 most common kinds of data provide a valuable test for whether the TDE is capable of expressing useful validation for the most common kinds of data, it could be asked whether they are entirely representative of the 3051 data columns not covered by these categories. A cursory inspection of these other kinds of data suggests that in many cases, they would be even easier to validate than the 32 most common kinds of data. Specifically, the majority of the tail categories appeared to match enumeration patterns, whose data descriptions are extremely easy to implement.

When implementing topes for each of the 32 most common categories in the spreadsheet data, 11 had enumeration patterns, 10 had word-like patterns, 7 had hierarchical patterns, and 4 had a mixture of hierarchical patterns for some formats and list-of-word patterns for other formats. (Section 3.4 described these common patterns.) None of these required numeric patterns, probably because the extract was intentionally biased toward string data.

Evaluating the value of expressing soft constraints

To evaluate how well these topes classified data as valid or invalid, 100 test strings were randomly selected for each category and validated with the corresponding tope implementations. After manually determining the validity of these test values, the accuracy of the topes was computed. F_1 is the standard machine learning statistic used to measure the accuracy of classifiers such as document classifiers and named entity classifiers, with typical F_1 scores in the range 0.7 to 1.0 [26][67].

$$Recall = \frac{\text{\# of invalid inputs successfully classified as invalid}}{\text{total \# of invalid inputs}} \qquad Precision = \frac{\text{\# of invalid inputs successfully classified as invalid}}{\text{\# of inputs classified as invalid}}$$

$$\mathrm{F_1} = \frac{2 \cdot \text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}$$

Five conditions were considered in order to evaluate the impact of identifying questionable inputs that could be double-checked:

Condition 1—Current spreadsheet practice: Spreadsheets allow any input. Since no invalid inputs are identified, recall and $F_1 = 0$.

Condition 2—Current web application practice: Since current spreadsheet practice is useless for identifying invalid inputs, I sought an alternate approach that would provide a reasonable basis for comparison. For this purpose, I turned to the current web application validation practices of professional programmers, who at least make some attempt to validate some inputs. In particular, as Section 2.4 discussed, programmers sometimes validate text fields, though only when it is convenient to find or create a regular expression for the field's data category or (for enumerable categories) when it is possible to require users to select inputs from a dropdown or radio button widget. Thus, I searched several online regular expression repositories and looked at the implementation of dozens of web forms in order to find regular expressions, dropdown widgets, and radio button widgets that could be used to validate the 32 test categories.

After several hours (more time than a programmer is likely to spend), this search produced 36 regular expressions covering 3 of the 32 categories (email, URL, and phone). In addition, based on the visible and internal values in dropdown widgets on the web, it was possible to construct 34 regular expressions covering 3 additional categories (state, country, and gender). Finally, assuming that programmers would use a checkbox for Boolean values yielded a regular expression covering the Boolean data category.

In short, the search yielded 71 regular expressions covering 7 of the 32 categories. While searching the web, it became clear that approximately half of the sites omitted validation *even for some of these 7 categories*. Nonetheless, the F_1 statistics (below) give the benefit of the doubt and are computed under the assumption that programmers would always validate these 7 categories and only accept inputs from the remaining 25 without validation.

Condition 3A—Tope rejecting questionable inputs: In this condition, the tope implementations tested each input s, accepting when $\phi(s) = 1$ and rejecting when $\phi(s) < 1$, thus making no use of topes' ability to identify questionable values. For comparability with Condition 2, the topes were restricted to a single format each.

Condition 3B—Tope accepting questionable inputs: In this condition, a single-format tope validation function tested each input s, which was then accepted when $\phi(s) > 0$ and rejecting when $\phi(s) = 0$.

Condition 4—Tope warning on questionable inputs: In this condition, a single-format tope validation function tested each input s, which was accepted when $\phi(s) = 1$ and rejected when $\phi(s) = 0$. If s was questionable, the process of asking a human to double-check s was simulated, meaning that s was accepted if it was truly valid or rejected if it was truly invalid. The advantage of Condition 4 is that it relies on human judgment in difficult cases when $0 < \phi(s) < 1$, thereby raising accuracy. The disadvantage is that a user would need to manually double-check each questionable input. Consequently, it is necessary to evaluate the tradeoff between increasing accuracy and burdening the user.[5]

[5] This experiment assumes that users are perfect "oracles" for double-checking strings. In practice, users are likely to make some errors when double-checking inputs, though no data are available for precisely determining the magnitude of this impact. Perhaps the most relevant numbers come from a study of users correcting the output of a named entity information extraction algorithm on Wikipedia data [43]. In this study, users were right in 80% of their judgments about whether the correct name entity strings were extracted by the algorithm. User errors were actually slightly biased toward *conservatism* (in that they rejected some strings that actually were correct). This Wikipedia study's task was actually harder than what users will need to do when double-checking questionable inputs identified by topes, since the experimental tasks required users to determine if the extracted string was not only a valid named entity, but also *the right* named entity. These results, particularly the users' conservatism, are encouraging, as they suggest that users generally would be able to identify most invalid strings in practice.

Results of single-format validation

As shown in Table 11, validating the 32 categories with single-format topes was more accurate than current practice (though well below the level of $F_1 = 0.7$ typically attained in many classifiers [26][67]).

Table 11. Asking the user for help with validating questionable inputs leads to higher accuracy in single-format validation

Condition	F_1
1 – Current spreadsheet practice	0.00
2 – Current web application practice	0.17
3A – Tope rejecting questionable values	0.32
3B – Tope accepting questionable values	0.32
4 – Tope warning on questionable values	0.36

Condition 2, current web application practice, was inaccurate partly because it accepted so many categories of data without validation. F_1 would likely be higher if programmers were in the habit of validating more fields. However, even in the 7 categories where programmers have published regular expressions on the web, or where it was possible to convert dropdown or radio button widgets to regular expressions, F_1 was only 0.31 (the same accuracy as Condition 4 in those categories), owing to a lack of regular expressions for unusual international formats that were present in the EUSES spreadsheet corpus. In other words, even in the 7 data categories where programmers do validate inputs, they do no better than a single-format tope. To achieve higher accuracy than topes, programmers would need to combine numerous international formats into a single regular expression for each data category, which stands in stark contrast to current practice.

Condition 4 improved accuracy to 0.36 versus 0.32 in conditions 3A and 3B by identifying questionable inputs and asking a human to double-check certain inputs. This 12% relative difference required asking the user for help on 4.1% of inputs. In other words, asking for a small amount of help yields a relatively large amount of increase in accuracy. The sensitivity is tunable: rejecting more inputs would reduce the burden on the user, at the cost of accuracy. For example, rejecting inputs that violate 2 or more production constraints, thus only asking about inputs that violate 1 constraint, would require asking for help on 1.7% of inputs but would reduce F_1 to 0.33. Rejecting inputs that violate any constraint would eliminate the need to ask about inputs, reducing to condition 3A, with F_1=0.32.

Results of multi-format validation

Evaluating the benefit of implementing multiple formats requires repeating the analysis above using more than one regular expression or format per category. In particular, the number of regular expressions or formats (N) was varied from 1 to 5. For example, with N=4, Condition 2 accepted each input `s` if `s` matched any of 4 regular expressions (selected from the 71 regular expressions). For each N, the performance of the best combination of N regular expressions is reported.

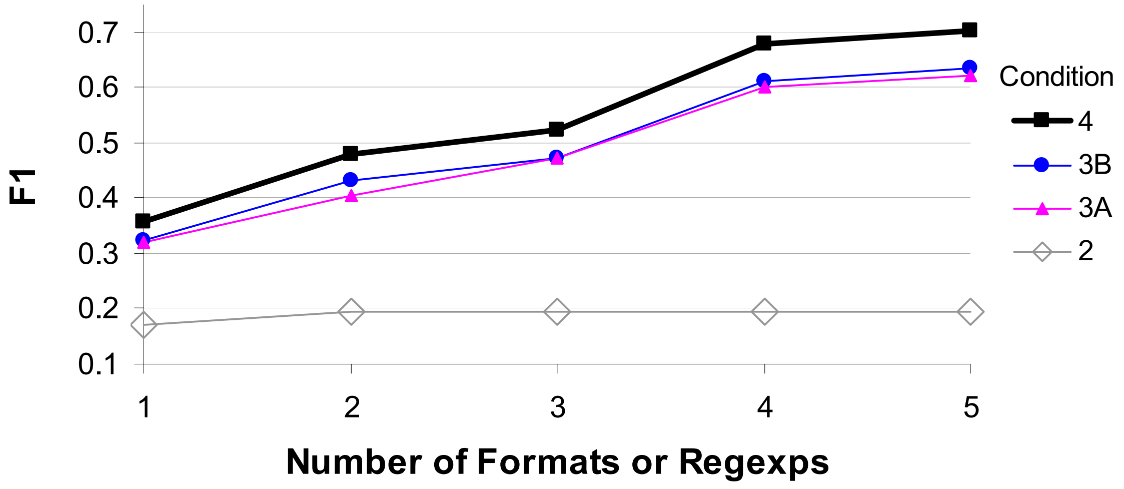
In Conditions 3A, 3B and 4, ϕ(`s`) = max(`isa`$_i$(`s`)), as discussed by Section 3.2, where i ranged from 1 through N. For each condition and each value of N, the performance of the best combination of N formats is reported.

Figure 25 shows that as N increased, so did accuracy. F_1 reached 0.7 when the number of formats reached 5.

Different regular expressions largely made the same mistakes as one another (such as omitting uncommon phone formats), so adding more than 3 regular expressions did not improve accuracy further. In contrast, increasing the number of formats continued to increase the tope accuracy, since the primitives supported by the TDE made it convenient to implement formats that are difficult to describe as regular expressions (such as expressing that a street name should contain a certain number of letters, no more than a few digits, and no more than a few words).

For each value of N, a 10% relative difference generally remained apparent between Conditions 3A/B and Condition 4, highlighting the benefit of identifying questionable inputs. In fact, as the number of formats increased, the number of questionable inputs decreased (as adding formats recognized more inputs as valid). By N=5, the 12% relative difference between Condition 4 and Conditions 3A/B required asking the user for help on only 3.1% of inputs.

Figure 25. Increasing validation accuracy by including additional formats



5.2.3 Usability for validating data

A between-subjects experiment assessed whether end users would receive benefits comparable to those described above. Lapis provided a suitable baseline for this experiment because, as discussed by Sections 2.7.3 and 2.7.4, Lapis includes support for inferring an editable pattern from examples and then identifying outliers that do not match the pattern. Empirical studies also have shown that end users are able to accurately read, write, and use Lapis patterns [70], which made it a more attractive baseline than regular expressions, which most end users do not know how to read or write.

The study was advertised through emails and posters (with emails sent to administrative assistants in the university roster, as in Section 2.2), recruiting 7 administrative assistants and 10 graduate students, who were predominantly master's students in information technology and engineering. None had prior experience with the TDE or Lapis, but many had some experience with programming or grammars. Each received $10 as compensation.

Participants were randomly assigned to a TDE or Lapis group. Each condition had four stages: a background questionnaire, a tutorial, three validation tasks, and a final questionnaire.

The tutorial introduced the assigned tool, coaching subjects through a practice task and showing all 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.

Task details

In Lapis, text appears on the screen's left side, while the pattern editor appears on the right. Users 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, Toped was embedded 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 (that is, $\phi(s) < 1$) or purple otherwise.

Each task presented 25 strings drawn from one spreadsheet column in the EUSES corpus [33]. Each column also contained at least 25 additional strings that were not shown 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. These data categories were selected to exercise the TDE 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".

Subjects were told 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, the instructions 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. These instructions were precise and made it possible to classify every single string as definitely valid or definitely invalid, which was essentially for comparability to Lapis, which is incapable of identifying questionable values.

Results

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

As shown in Table 12, the conservative Mann-Whitney (Wilcoxon) statistical test was used to assess results, since the sample was small and not necessarily normally distributed data (though all measures' medians were very close to the respective means).

Table 12. Results comparing TDE to Lapis.

	TDE	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

In the allotted time, TDE subjects completed an average of 2.79 tasks, while Lapis subjects averaged 1.75 (Table 12), a significant difference ($p<0.01$). TDE users were more successful at their primary goal, finding typos. Of the 18 actual invalid strings in the 75 visible strings, TDE subjects found an average of 16.5 invalid strings, compared to 5.75 for Lapis subjects, which was a significant difference ($p<0.01$). In addition, of the 35 typos in the total set of 150 test strings, the completed TDE formats found an average of 31.25 invalid strings, whereas completed Lapis patterns found only 9.5, a significant difference ($p<0.01$).

Finding invalid data is not sufficient alone. Validation should also classify valid data as valid. As in Section 5.2.2, the standard F_1 statistic was used [26]. The 23 completed TDE formats had an F_1 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.

Thus, the measured F_1 accuracy scores for the TDE are higher than the measured F_1 accuracy scores for Lapis (though the inter-tool differences are not statistically significant at $p<0.05$). In fact, these TDE scores are comparable to those demonstrated by Section 5.2.2, showing that the TDE is highly effective at helping actual end users to create abstractions for validating strings.

Subjects had different job categories and varying experience with grammars and programming. Yet for each tool, there were no statistically significant effects ($p<0.05$) on task completion, format accuracy, or user satisfaction based on this prior experience or job category.

The study closed with a user satisfaction questionnaire because users 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 [78].

Subjects generally commented that the TDE 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 (as was the case in the old Toped tool), which helped to prompt the new structure reflected in the Toped^{++} user interface.

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. Answers had moderate internal consistency (Cronbach's alpha=0.74) and were combined into a scale. On this scale, subjects reported an average satisfaction of 3.78 with the TDE and 3.06 with Lapis, a significant difference ($p=0.02$).

Comparison to regular expressions

Though not perfectly comparable, it appears that subjects completed these tasks with the TDE more quickly and accurately than subjects completed tasks with regular expressions in a study during SWYN's development [16]. 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 regular expression 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, the TDE 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 an error rate of only 19%. Statistical tests between the two studies would be meaningless, due to the differences in the task details, but these results suggest that just as the TDE offers a significant improvement over Lapis, it also offers a substantive improvement over regular expressions when placed in the hands of end users.

Chapter 6. Transformation functions

When users combine data from multiple sources into a data set such as a spreadsheet, the result is often a mishmash of different formats, since instances of a string-like data category typically can appear in a variety of formats. For example, a phone number copied from one web site might be formatted like the string "393-202-3030", while a phone number copied from another web site might be formatted like "(262) 444-5000". Putting instances of each data category into a consistent format then requires reformatting some or all of the instances. Although spreadsheet-editing applications like Excel provide features for reformatting numbers and a few specific kinds of string data, they do not provide any support for reformatting the wide range of other kinds of string data encountered by users. Likewise, web forms and web macro tools provide no support at all for automatically reformatting data. As a result, users must manually perform these reformatting operations, which is tedious and error-prone.

As described in Chapter 3, a tope can include functions that each reformat strings from a source format to a target format. Once implemented, the TDE can pass strings into these functions as needed, thereby enabling the TDE to reformat strings automatically. The latest TDE prototype automatically implements reformatting functions based on the data description created by the user in Toped^{++}.

As with format validation functions, there are two key questions related to reformatting functions. First, does the TDE provide enough expressiveness to accomplish the job (in this case to reformat data in a way that supports user tasks)? Second, does the TDE enable end users to accomplish useful work (in this case, to reformat strings)? After Section 6.1 presents features in the TDE related to reformatting, Section 6.2 presents two studies that answer these questions. The first study, described by Section 6.2.1, shows that the TDE is sufficiently expressive for reformatting Hurricane Katrina web site data in order to help identify duplicate values; as explained by Section 2.4, this reformatting had required a huge amount of manual labor. The second study, described by Section 6.2.2 shows that 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 strings in this format in future spreadsheets, as described in Chapter 7.

6.1 TDE support for generating and calling transformation functions

The TDE automatically implements `trf` functions based on the data descriptions created by users for data categories. When reformatting a string at runtime, the TDE checks whether ambiguity might cause the TDE to reformat the string incorrectly, then displays an overridable warning message if needed so that the user can double-check and possibly correct the result.

6.1.1 Generating transformation functions

As with format validation functions, the Topeg module can automatically generate implementations of `trf` functions. Previous versions of the system (described by Section 6.1.3) required users to manually implement `trf` functions. Topeg implements these functions based on the data descriptions created in Toped^{++}.

It is most straightforward to first describe the implementation of simple functions for a data description containing a single variation with a single individual Word-like and Numeric parts, then explain how Topeg implements `trf` functions for data descriptions that have more structure.

Data description containing a single variation with a single Word-like part

If a data description 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 18) and the formats of the data description. Specifically, the user may indicate that the part can be in upper case, lower case, title case, or mixed case formats.

For each pair of formats `(x, y)` in a data description with a single variation containing a single Word-like part, Toped^{++} automatically implements a function $\texttt{trf}_{xy}$ that reformats inputs into the capitalization required by format `y`. For example, if `y` is the upper case format, then each $\texttt{trf}_{xy}$ converts each character to upper case. If `y` is the title case format, then $\texttt{trf}_{xy}$ title-cases each word except for those in the exception list. (As an exception to the exception, English prepositions are 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 18.) If `y` is the mixed case format, then $\texttt{trf}_{xy}$ simply returns the input unaltered. Thus, without any explicit effort by the user, Topeg 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 18), then Topeg adds or removes `trf` functions accordingly.

Data description containing a single variation with a single Numeric part

Topeg handles numeric parts 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, Topeg generates a $\texttt{trf}_{xy}\texttt{(s)}$ that rounds `s` (or appends '0' characters) so the output has the number of decimal digits required by `y`. As with a data description 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.

Data description containing a single variation with a single hierarchical part

A hierarchical part matches some other data description `t`. Thus, if a data description `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`, Topeg implements a function $\texttt{trf}_{xy}$ 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 respective data description.

Data description containing than one part or variation

The discussion above explained how Topeg implements `trf` functions for single-variation, single-part data descriptions. In general, data descriptions have multiple variations and multiple parts.

Suppose that a data description has formats `x` and `y`. In particular, let `y` contain separators $<s_0, s_1, \ldots, s_m>$ and parts $<y_1, y_2, \ldots, y_m>$ in formats $<f_1, f_2, \ldots, f_m>$. (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 26.

Figure 26. Implementation of $\mathtt{trf}_{xy}$

```
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
```

Note that the `trf` implementation in Figure 26 reformats each part y_k to format f_k. This is achieved by treating y_k as a single-variation, single-part data description, one of whose formats is f_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, Topeg treats it as a single-variation data description with a single Word-like part and calls the `trf` function described above to put that part into title case.

The automatically-generated transformation functions have a number of limitations explored in Section 6.1.3.

Supporting transformation based on whitelist entries

The automatically-generated reformatting rules typically suffice for most data reformatting tasks, since the formats in most data categories are usually related to one another via straightforward permutations of parts, changes of separators, changes of capitalization, and padding or rounding of numbers. However, some categories such as building names (as in Figure 1) call for a fixed list of allowed values, each of which might have a synonym such as an abbreviation.

Sometimes, these synonyms do not match any of the usual formats for that kind of data. For example, in Pittsburgh, "411" is a valid phone number equivalent to "(412) 555-1212", which provides directory assistance. To accommodate these situations, the TDE supports whitelists. The "whitelist" tab of Figure 10 provides a grid where the user can enter a column of values that match the main data description, as well as synonyms (Figure 21).

Internally, Topeg creates a new format x for each synonym column; $\mathtt{isa}_x$ returns 1 for any string that matches a value in the column, or 0 otherwise. For each pair (x_1, x_2) of whitelist formats, Topeg implements a $\mathtt{trf}_{x1x2}\mathtt{(s)}$ 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.

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

Supporting custom JavaScript-based transformation functions

While automatically-generated reformatting functions suffice for virtually all data categories, the TDE supports full Turing-completeness by allowing users to override automatically generated `trf` functions by writing custom code. This is helpful when transforming a string from one format to another requires a complicated computation in order to determine what the value of a certain part should be. For example, converting an ISBN from 10 digits to 13 digits involves truncating the last digit, prepending "978", calculating a check digit (using modulo arithmetic), and appending this check digit to the string. Implementing a function in JavaScript requires turning on a custom view (under the "View" menu of Figure 10) and drawing arcs from one variation another. For each arc, the user can specify JavaScript instructions that reformat and concatenate the parts of the variation. These instructions are called at runtime instead of automatically implemented functions. In practice, JavaScript-based functions are almost never needed. In particular, none of the data categories involved in any studies actually called for JavaScript.

6.1.2 Limitations of transformation functions in the TDE

Although the TDE's support for automatically implementing transformation functions saves a great deal of work, since it is rarely necessary to write JavaScript, the approach overall has three primary limitations. First, transformation functions sometimes produce invalid outputs. Second, transformation functions sometimes destroy information, meaning that the user is unable to apply the inverse transformation to automatically recover the original string. Finally, ambiguity about which transformation function to apply can sometimes cause the TDE to reformat strings in a way that is not intended by the user. In practice, these limitations rarely come into play, and the TDE also provides several features to help users identify situations where these limitations might affect the results of a transformation.

Applying transformations can generate invalid outputs

While the automated transformations generally work well, they can sometimes produce erroneous outputs. Almost always, this is because the target format contains parts that the source format does not (so the transformation has no way of determining how to fill in the missing part during the transformation). For instance, in Figure 18, every person name 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 $\mathtt{trf}_{xy}$ might not match $\mathtt{y}$.

To detect errors of this kind, the TDE always checks the output of the transformation with the grammar for the target format, yielding an error message prompting the end user who provided the input to review and manually correct the output.

Reversing transformations automatically can be impossible

For any pair of formats $\mathtt{x}$ and $\mathtt{y}$ and any string $\mathtt{s}$ such that $\mathtt{isa}_{x}\mathtt{(s)} = 1$, $\mathtt{trf}_{yx}\mathtt{(trf}_{xy}\mathtt{(s))}$ generally equals $\mathtt{s}$. In other words, the automatically generated transformations generally behave as inverses of one another when applied to valid strings. The reason is that most strings in most spreadsheets or web forms are valid, and most transformations involve just capitalization changes (which are invertible for most common strings), permutation (which is perfectly invertible), separator changes (again perfectly invertible), and lookup tables (which are almost always invertible). Because $\mathtt{trf}_{yx}$ and $\mathtt{trf}_{xy}$ generally behave as inverses of one another, a user can apply $\mathtt{trf}_{yx}$ to transform $\mathtt{trf}_{xy}\mathtt{(s)}$ back into $\mathtt{s}$.

However, many of the transformation operations described in Section 6.1.1 are not perfectly invertible. First, even with a single Numeric part, rounding destroys information and prevents recovering the original string through the reverse operation (padding with zeroes). Second, even with a single Word-like part, capitalization can destroy information when changing from the mixed case format to any other format (e.g.: converting "McCain" to lowercase yields "mccain", which is left as "mccain" by the function that transforms to mixed case). Third, transformation within a single Hierarchical part or between two variations can drop parts when transitioning from one format to another (as when converting "Christopher P. Scaffidi" to "Scaffidi, Christopher"). Finally, lookup tables theoretically could destroy information if they map multiple strings in one format to a single string in another format (e.g.: mapping both of the airport codes "SFO" and "OAK" to "San Francisco").

Of these four forms of information loss, the third one (dropping of parts between variations) seems to occur most often, probably because users commonly work with a handful of data categories whose strings are sometimes written with missing parts. The principal examples are person names, phone numbers (where the area code is sometimes dropped), and URLs (where the leading protocol is sometimes dropped if it is "http").

It should be noted that even with these forms of information loss, the automatically-generated transformation functions *perform no worse* than a user would if he performed the same transformations. For example, regardless of whether a tool automatically rounded a number or a human manually did it, the information is still gone: there is no way to inspect the resulting number and determine what number it came from. Thus, the TDE's support for automation does no harm to users' ability to invert transformation operations—and in most cases, it does a great deal of good, by saving users the time and attention required to perform so many tedious and error-prone operations manually.

Choosing the right transformation can be ambiguous

The TDE's approach for implementing reformatting rules yields a totally-connected tope graph, so there is no need for "chaining" reformatting functions. Instead, for any two formats x and y, there is always a direct route through the tope graph. When reformatting strings, the TDE always takes this direct route, thereby addressing one form of ambiguity identified by Section 3.2: the question of which route to take through the tope graph.

However, to actually transform a string s at runtime to a format y selected by a user, it is necessary to first determine the *starting* format x of s so that the TDE can call the appropriate trf_{xy}. To achieve this, the TDE parses s with the subsuming grammar generated by Topeg (described by Section 5.1) to determine what format is the best match to s. The TDE chooses the format x with the maximal isa score as the starting format.

However, this raises the 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 $isa_{x'}(s)=isa_x(s)$ but $trf_{x'y}(s)\neq trf_{xy}(s)$. For example, Americans might interpret "04/01/2009" as April Fool's Day, but Europeans might not immediately interpret the string this way because their preferred date formats generally place the day before the month. Formatting the string "04/01/2009" to the international standard date notation (ISO 8601) could produce "2009-04-01" or "2009-01-04", depending on which continent's preferred format is used as the source format. It would be ideal to compute $trf_{x'y}(s)$ and compare it to $trf_{xy}(s)$ 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 a couple.

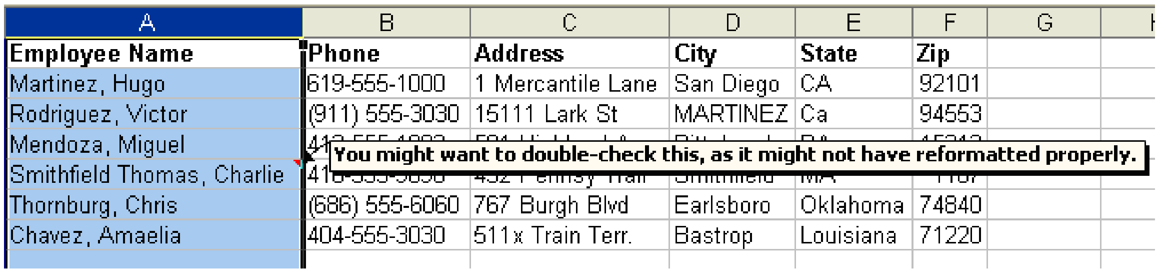
Nonetheless, some x' can be ruled out, in the cases where $trf_{x'y}(s)$ could not possibly differ from $trf_{xy}(s)$. Specifically, ambiguity at the Word-like part and Numeric part levels can be disregarded. For example, regardless of whether s is interpreted as upper case or title case, its lower case 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 is "2010-10-10". "04-01-08", on the other hand, is problematically ambiguous precisely because different formats match different substrings to month, day, and year.

Because 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 randomly selected data values used in the studies described by Section 6.2 happened to have reformatting problems caused by ambiguity. 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, the TDE automatically detects when ambiguity might have caused an incorrect reformatting, and it presents a warning message that the user can override after double-checking the string. Specifically, if parsing a string `s` with a subsuming grammar yields more than one parse tree, then the TDE checks whether any of these trees match different literal substrings of `s` to the parts of the data description. If even a single part is assigned different strings by different parses, then the ambiguity is problematic and could an unintended output, so the TDE displays the overridable warning message, "You might want to double-check this, as it might not have reformatted properly." For example, if a person's first and last name could each contain more than one word, then "Smithfield" in "Charlie Smithfield Thomas" could be part of the first or last name. In this case, permuting the names when reformatting "Charlie Smithfield Thomas" would involve problematic ambiguity, resulting in a warning message (Figure 27).

Figure 27. Displaying a warning message due to ambiguity during reformatting.



A	B	C	D	E	F	G
Employee Name	**Phone**	**Address**	**City**	**State**	**Zip**	
Martinez, Hugo	619-555-1000	1 Mercantile Lane	San Diego	CA	92101	
Rodriguez, Victor	(911) 555-3030	15111 Lark St	MARTINEZ	Ca	94553	
Mendoza, Miguel	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	
Smithfield Thomas, Charlie	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	
Thornburg, Chris	(686) 555-6060	767 Burgh Blvd	Earlsboro	Oklahoma	74840	
Chavez, Amaelia	404-555-3030	511x Train Terr.	Bastrop	Louisiana	71220	

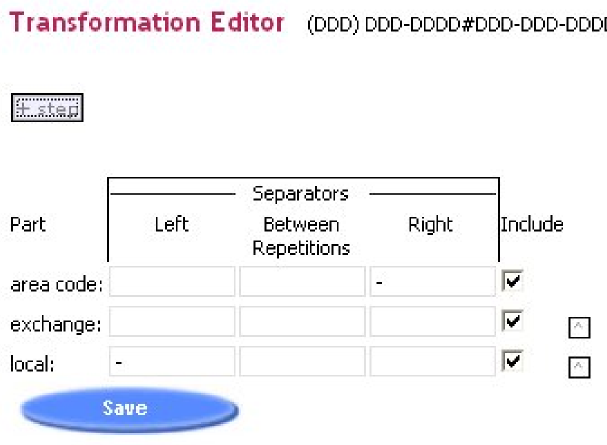
You might want to double-check this, as it might not have reformatted properly.

6.1.3 Implementing transformation rules with the early Toped prototype

Like the Toped format description editor (described by Section 5.1.3), Toped's transformation editor is forms-based. The user interface has two sections (Figure 28). The first section shows a list of steps required for reformatting each part of the source format. Clicking the "+ step" button allows the user to select from three kinds of primitives: a primitive to change the capitalization, a primitive to replace a part's substring using a lookup table, and a primitive to call a `trf` function in another tope (which is useful when the part is defined to match a format in another tope). A transformation description for phone numbers does not require any of these primitives, but just for illustration, Figure 29 shows what a lookup table for state names looks like. This would be useful, for example, if a user needed a tope for reformatting the last line of an address from a string like "PITTSBURGH, Pennsylvania" to a string like "Pittsburgh, PA".

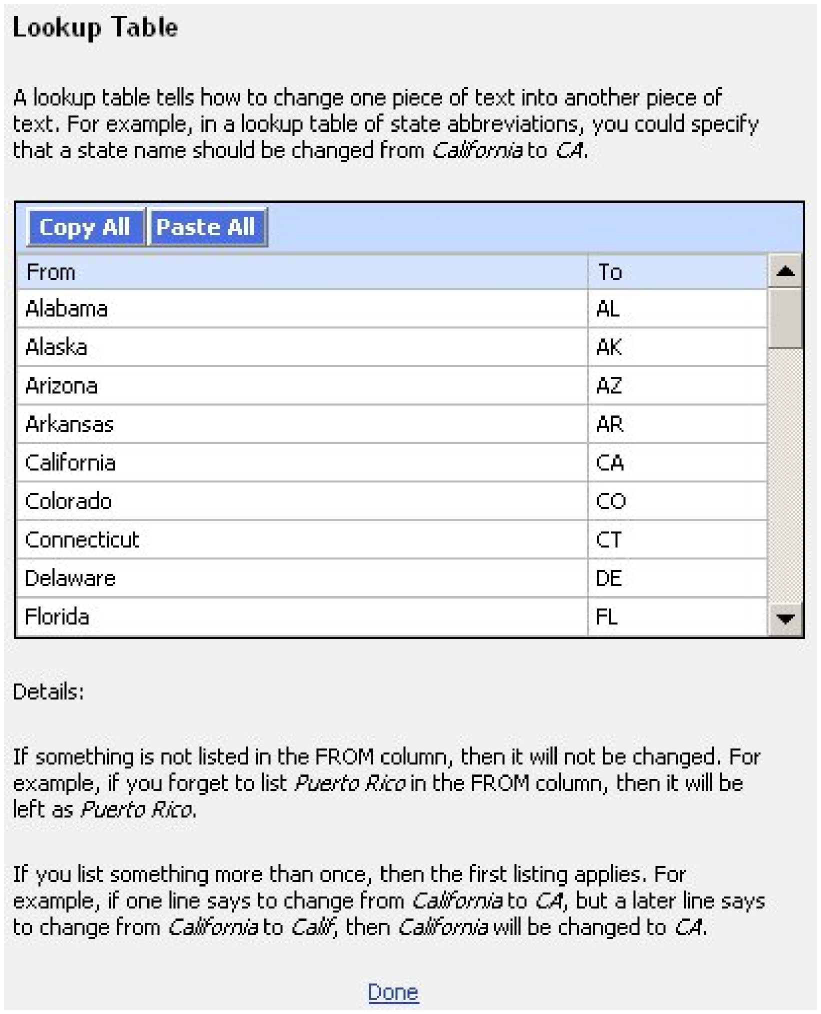
The second section of the editor shows what separators to use when concatenating reformatted parts to produce a string in the target format. The user can specify a new separator to the left and right of each part. In addition, because a format can specify that parts may repeat (with separators between the parts), the editor allows the user to specify what the inter-repetition separators should become during concatenation. The editor supports permutation of parts (with the "^" buttons) and omission of parts (by unchecking the "Include" checkbox) though these features are not required for phone numbers.

Figure 28. Editing a `trf` function in Toped for reformatting phone numbers



Part	Separators Left	Separators Between Repetitions	Separators Right	Include
area code:			-	✓
exchange:				✓
local:	-			✓

Figure 29. Editing a lookup table in Toped for reformatting from state name to state abbreviation.



Lookup Table

A lookup table tells how to change one piece of text into another piece of text. For example, in a lookup table of state abbreviations, you could specify that a state name should be changed from *California* to *CA*.

From	To
Alabama	AL
Alaska	AK
Arizona	AZ
Arkansas	AR
California	CA
Colorado	CO
Connecticut	CT
Delaware	DE
Florida	FL

Details:

If something is not listed in the FROM column, then it will not be changed. For example, if you forget to list *Puerto Rico* in the FROM column, then it will be left as *Puerto Rico*.

If you list something more than once, then the first listing applies. For example, if one line says to change from *California* to *CA*, but a later line says to change from *California* to *Calif*, then *California* will be changed to *CA*.

Data descriptions created in Toped are somewhat brittle, in that modifying formats often breaks the transformations that join one format to another. For example, if the user adds a new part to a format, then any transformation rules sourced from or targeted to that format must be updated. Toped^{++} automatically implements reformatting rules and repairs them as needed as users add, edit, or delete parts. By performing this work automatically, Toped^{++} greatly reduces the time and effort required to create and maintain multi-format data descriptions. The automatically-generated reformatting rules typically suffice for most data reformatting tasks. In the rare situations requiring complex reformatting, users who know JavaScript can override automatically generated `trf` functions, as described by Section 6.1.1.

6.2 Evaluation

Since the primary goal in this research is to help end users reformat short, human-readable strings, one of the most important evaluation criteria is whether the TDE enables end users to create and use topes to reformat inputs. As with evaluating the TDE's support for validation (in Section 5.2), evaluating support for reformatting strings has two questions: does the system provide adequate functionality for expressing reformatting, and does this functionality enable end users to complete tasks more quickly and effectively than through the currently practiced techniques?

First, is it even possible at all to express useful reformatting operations in the TDE? This question leaves aside the issue of usability by end users and focuses on the intrinsic capability of the system. In Section 6.2.1, expressiveness is evaluated by using the primitives of the Toped-based TDE to implement reformatting operations for data obtained from a Hurricane Katrina "person locator" site, then measuring whether applying those operations to the data would have helped identify duplicate values. While this procedure only identified 8% more duplicate values, the process was relatively straightforward and fully automated (after creating the needed topes), which would have saved a great deal of the manual labor described by Section 2.4.

The second stage of this evaluation addresses whether putting the TDE in the hands of end users actually enables them to quickly and effectively reformat strings. A within-subjects laboratory experiment described by Section 6.2.2 answers this question by having some users reformat spreadsheet data with the Toped^{++}-based TDE and then perform the same tasks manually (since existing end-user applications currently require people to perform string reformatting by hand). The results show that, 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". These results strongly suggest that the TDE enables users to reformat strings much more quickly and effectively than is possible with current approaches.

6.2.1 Expressiveness for reformatting to find duplicate values

After the proliferation of Hurricane Katrina "person locator" sites (Section 2.4), some teams of developers attempted to combine the sites into unified databases. Since some hurricane survivors were mentioned in records on multiple web sites, a straightforward aggregation of the data led to many duplicate values. Unfortunately, since different teams independently developed different sites, their data often had different formats.

Consequently, putting data into a consistent format was a necessary precondition to identifying duplicates. Having put strings into a consistent format, the workers could find duplicate strings, and then could identify duplicate records (person name + phone number + address information) based on duplicate strings. Due to the large number of sites to be combined and the fact that most volunteers on the aggregation project lacked skills for writing scripts, this reformatting was done manually for the most part. Ideally, it should be possible to automate the reformatting actions by expressing the necessary reformatting rules with the TDE.

Thus, an important test for the effectiveness of the TDE is to check whether it is possible to express suitable reformatting functions for putting strings into a consistent format. One measure of this effectiveness is to see how many new duplicate strings are uncovered. (Another possible measure would be to see how many duplicate records could be found based on duplicate strings, but this is a less suitable measure of the TDE's effectiveness because it would conflate whatever record-matching algorithm was used with an evaluation of reformatting.)

To generate test data, a script downloaded the web pages from one person locator site[6] that had 5 text fields: person last name, first name, phone, address line, and city. (It also had state names, but this was not an appropriate kind of data for this evaluation, since the site provided a dropdown field for users to select a state in a consistent format.) The script randomly extracted 10000 values for each of the five categories.

As described by Section 5.2.2, 32 topes had been created in the Toped-based TDE to validate spreadsheet data. Five of these served as a starting point for reformatting instances of the corresponding data categories in the Hurricane Katrina data. For each tope, the best five formats (identified by Section 5.2.2) were retained without modification and used to validate the Hurricane Katrina data, thereby identifying the most commonly-occurring format for each data category in the Hurricane Katrina data. Reformatting functions were manually created to put strings into the format that occurred most often.

[6] http://www.publicpeoplelocator.com/

Finally, after putting strings into their category's most common format, the number of duplicate strings were counted and compared to the number of duplicates prior to reformatting. Reformatting produced approximately 8% more duplicates after transformation. The person first and last names categories benefited the most (with 16% and 9% additional duplicates, respectively), largely because users often entered middle initials into these fields. Each of the matching topes had a format that accepted a name and an initial, and the reformatting removed the initial, thereby enabling duplicate detection. The address line data (with 9% additional duplicates) largely benefited because of replacing abbreviations such as "Ave." with equivalents such as "Avenue". There were a moderate number of new phone duplicates (8%). There were few additional city duplicates (2%) because almost all inputs contained "New Orleans", leaving only a few new duplicates to be discovered (largely due to replacing uppercase city names with title case names). It has also been suggested that additional duplicates could be found based on incorporating abbreviations for city names into that data description. Regardless, it seems likely that topes would be even more effective on a more diverse city dataset.

Perhaps the number of new duplicates (8%) might seem small. However, the main benefit here is that a task was automated that had not been effectively automated by Hurricane Katrina volunteers. Thus, it was possible to perform in minutes what literally required hundreds of man-hours of labor. The next question, addressed by Section 6.2.2, is if end users would also reap these benefits in practice.

6.2.2 Usability for reformatting data

A within-subjects experiment assessed the usability of the Toped^{++}-based TDE for helping users to create topes and to reformat spreadsheet data. As a baseline, this experiment compared using the TDE to how long it would take each user to perform the same tasks by manually typing changes into Excel.

The study was advertised by emails and posters to recruit 9 master's students, who were predominantly in the school of business. None had prior experience with the TDE. To filter out users who already understood the concept of designing string recognizers, participants were screened to ensure that none had experience with regular expressions.

The experiment had four stages: a tutorial, three tasks with the TDE, the same tasks but without the TDE, and a user satisfaction questionnaire. Because the study did not counter-balance users (that is, giving some participants the manual tasks prior to the TDE tasks), users were more familiar with their goals when they started the manual tasks than when they started the tasks with the TDE. Consequently, this experiment probably measured *lower bounds* on the relative benefits of the TDE. Users could take up to 30 minutes for the tasks in the TDE (though all participants finished sooner), but they could only spend 60 seconds on each of the 3 manual tasks. This time limit was chosen with the intention to avoid wearying participants with tedium yet to allow measuring the average number of seconds needed 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 then to reformat all cells to a specific alternate format. For example, one task required users to find typos in a spreadsheet column of email addresses, some of which were missing a user name or a ".com". Some cells had an extra email address. Participants had 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. This organization placed data categories into groups based on similarity: the first group contained single-word single-variation data categories, the second group contained data categories whose instances could be reformatted using a whitelist, and the third group contained multi-part multi-variation data categories. This organization was motivated by the expectation that most users would not be able to master all of the features of the TDE within a short tutorial.

For whitelist-based tasks, participants were provided with a table in Microsoft Word, which they could copy-and-paste into the Toped^{++} whitelist editor. These tables were copied from the web; for example, the list of US states was copied from Wikipedia. This saved participants the trouble of searching the web to find appropriate tables. However, this does not bias the study in favor of the TDE, 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 data categories were among the most common identified in the earlier analysis of how well topes could validate the EUSES corpus of spreadsheets [33] (Section 5.2.2). For each kind of data, 100 strings were randomly selected from one spreadsheet column in the corpus.

Speed and accuracy

Using the TDE, 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, implementing 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 it seemed to be a matter of misunderstanding instructions rather than a usability problem in the tools.)

Different participants took different approaches to manually perform each task. In 14 out of the 27 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 by each user, four statistics were 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, these statistics sufficed for computing the average number of seconds to fix each typo and average number of seconds required to reformat each cell. (The two cases where users gave up were omitted from this computation.) Combining this with the rate of typos in the columns yielded projections of how long it would take participants to manually fix every typo in each 100-cell column, which ranged from 5.0 to 15.9 minutes (Table 13). As when using the TDE, fixing typos in whitelist data (Group 2) required repeatedly referring to the list in Word, which was relatively slow. Moreover, whereas the TDE 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.

The statistics also provided enough information to compute 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".

Table 13. Comparing time required to fix typos and reformat 100-cell columns

	Minutes Required		Breakeven
	TDE (actual)	Manual (projected)	point (# 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**

User satisfaction

The user satisfaction questionnaire asked each participant if he preferred using the new Excel feature or doing the tasks manually. Every participant indicated a strong preference for the TDE. In addition, the questionnaire asked users to rate on 5-point Likert scale how hard the TDE 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 satisfied end of the scale). Moreover, two people verbally launched into detailed descriptions of how they wished that a tool like this had been available in office environments where they previously had worked, in order to perform these kinds of reformatting tasks. These results suggest that users would eagerly adopt the TDE if it were made widely available.

Chapter 7. Reuse of tope implementations

Successful communication requires people to have a common model for interpreting the signals that they exchange. For example, if someone speaks Spanish, then only people who know that language can understand the speaker's words. As the speaker converts messages into sentences composed of words and converts words into spoken sounds, the listeners rely on their knowledge of the language to interpret those sounds and understand the speaker's message. Essentially, speaking is an encoding process, while listening is a decoding process. It is not necessarily the case that the speaker and the listener must have precisely the same language model, but their two models must have enough in common to reduce the risk of misinterpretation to a manageable level.

In a similar fashion, people must have a (mental) tope in common in order to communicate the short, human-readable strings in data categories discussed by this dissertation. For example, if one person writes a US postal address, another person relies on familiarity with that kind of data in order to understand the written string. The writer and the reader each call upon a common mental model to make this communication successful. The writer relies on this mental model in order to format and concatenate the parts of a mailing address into a series of characters, while the reader relies on the mental model to interpret the series of characters. This shared mental model corresponds to one tope abstraction. The reader and the writer have the same tope—or, if their topes differ a bit, they must have enough in common that the reader and the writer experience no more than a manageable level of miscommunication.

Since communicating such strings requires people to have a tope in common, it is unsurprising that few topes are unique to individual people. Instead, topes are generally used throughout a society or an organization. US postal addresses are an example of a society-wide tope; university accounting codes are an example of an organization-wide tope uncovered in the Carnegie Mellon contextual inquiry (Section 2.3). While not every string is necessarily communicated between people, enough instances of these data categories are communicated often enough that many people benefit from knowing topes for these data categories. For example, when a new employee joins Carnegie Mellon University, he may not immediately know how to read and write university accounting codes, but because other people use those codes, there is a value to learning a tope for that data from other employees.

Likewise, if one citizen in a society or one employee in an organization finds it useful to create a tope implementation so that his computer can automatically recognize and reformat instances of a data category, then other citizens in the same society or other employees in the same organization might also benefit from using that tope implementation. For example, if one employee creates a tope implementation to validate university accounting codes in a spreadsheet, then another employee might also benefit from using that implementation to validate accounting codes in another spreadsheet. Even if the two employees are working with completely different spreadsheets containing different actual strings, the same tope implementation might be useful to both employees because they have an organization in common and thus a tope in common.

Moreover, because data categories are generally tied to societies or organizations, they are generally independent of particular software applications. For example, it would be very annoying if university's staff had to use one system of accounting codes whenever they worked in Microsoft Excel, an entirely different system of accounting codes whenever they worked in Oracle databases, and a third entirely different system of accounting codes on web pages. In this hypothetical example, there would be three different data categories. Such a situation is nonsensical: what kind of an organization would (willingly use a different data category for essentially the same purpose every time its users moved from one application to another? Thus, data categories tend to be application-agnostic. If a person needs to validate or reformat instances of a particular data category in more than one application, it would be ideal if he could use the same tope implementation in each application.

Reuse of tope implementations could have a number of benefits. First, it could be faster for people to reuse an existing implementation, rather than creating one from scratch, thereby saving people time. Second, if a certain person (such as a new employee) is not yet familiar with a certain kind of data, then it might be impossible for him to implement a tope for that data, so reusing an existing tope implementation might be the only way to acquire one. Third, if different people reuse the same tope implementation, then their computers would validate and reformat the corresponding data with the same rules, which might help to reduce inconsistency and possibly the potential for miscommunication; likewise, reusing the same tope implementation to validate a certain kind of data in multiple applications could help to ensure consistency among those applications. Fourth, some people in an organization (such as the system administrators, or highly-skilled end users sometimes called "gardeners" [78]) are more likely than other people (such as administrative assistants) to have the skills needed to design high-quality tope implementations, particularly for especially complex topes. For example, a system administrator would probably be more cognizant of the fact that the top-level domain part of a URL should fall into a certain set that includes a large number of two-character country codes. If all workers could reuse these good tope implementations, then they would not need to create potentially-buggy tope implementations on their own. Fifth, regardless of the skill level of the person who uses an existing tope implementation, just having it on hand might help to speed the creation of new tope implementations, since he could refer to it as an example of how to implement a tope, and he might even be able to incorporate it into a larger tope implementation (as when reusing a person name tope implementation and an email address tope implementation to validate strings like "Christopher Scaffidi <cscaffid@cs.cmu.edu>").

In short, reuse is feasible because many topes are not unique to a single application or to a single person. Reuse is desirable because it could save users time, reduce errors, and help ensure consistency.

In the TDE, reusability of tope implementations is predicated on a simple two-part strategy: factor out all tope-specific concerns into an *application-agnostic* tope implementation that can be stored on a users' computer or a server, then *recommend* particular tope implementations based on keywords and examples of the data to be validated and reformatted. Therefore, the most crucial evaluation criteria for reusability are whether tope implementations really are application-agnostic, and whether the TDE actually can accurately recommend appropriate tope implementations in a reasonable amount of time.

As the first part of the reuse strategy, the TDE separates tope-specific concerns from application-specific concerns by providing a programmatic API called by applications via add-ins. Each add-in passes strings from the application into the TDE via the API, then displays return values on-screen in an application-appropriate manner. The API effectively acts as a partition between application-specific concerns handled in the add-ins and application-independent concerns handled in the tope implementations. Section 7.1.1 describes this API in more detail, with a particular focus on what specific steps are required to implement a new add-in. Section 7.2.1 evaluates the cross-application reusability of tope implementations, demonstrating that they can be reused in an application-agnostic fashion without loss of accuracy in both spreadsheets and web forms.

As the second part of the reuse strategy, the TDE recommends tope implementations for reuse through a "search-by-match" algorithm described by Section 7.1.2. This algorithm recommends tope implementations based on how well they match example strings provided by the user. For example, this algorithm could quickly locate tope implementations that match "888-555-1212". The algorithm can be used to search a user's personal computer for topes; in addition, as described by Section 7.1.3, the algorithm has been incorporated into repository software that allows users to upload, search, and download data descriptions. Section 7.2.2 shows that this algorithm has a recall of over 80% and a query time of under 1 second when searching, making the algorithm accurate enough and fast enough to observe data being edited by a user and to search the user's computer for a tope implementation for that data. Section 7.2.2 concludes by describing how the search-by-match algorithm could be parallelized in order to search through large repositories.

7.1 TDE support for tope reuse

Suppose that Bob wants to validate and reformat some data in a spreadsheet. Perhaps a suitable tope implementation was already created by Alice and stored on a repository server. Or perhaps a suitable tope implementation was already created by Bob and stored on his own machine. In either scenario, it is necessary to get the string data from the spreadsheet into the tope implementation's `isa` and `trf` functions, then use the functions' return values to update the spreadsheet application.

To support the first scenario, with the tope implementation on a repository, the TDE provides a user interface so that Bob can search through the repository and find a desired tope implementation (Figure 16). Architecturally, this user interface runs on Bob's computer as a client to the repository, which is a server. When Bob clicks on a button for the desired tope implementation, the client downloads the selected tope implementation's data description and stores a copy locally. A tope implementation is now located on the same machine as the spreadsheet, just as if Bob had created it himself, thereby reducing the first scenario to the second.

In the second scenario, the needed tope implementation is an old one already on the local machine—either because Bob created it himself, or because he already downloaded it. For these situations, the TDE provides a user interface so Bob can enter keywords and example strings, which the TDE uses to search through the tope implementations stored on the local computer. Individual add-ins can provide custom user interfaces as well (Figure 15). Internally, these user interfaces rely on a copy of the repository software running locally, communicating via memory with the user interface, rather than via a network connection (as in the case of browsing a repository server).

Once Bob has selected a tope implementation that is stored on the local computer, it is still necessary to get the strings from the spreadsheet into the tope implementation, then display the functions' results to the user. To achieve this, the TDE provides a programmatic API called by applications via add-ins. Each add-in passes data values into the TDE via the API, then displays return values on-screen in an application-appropriate manner. For example, the Excel add-in passes spreadsheet cells' strings through the API into the TDE, which parses the cells using the grammars generated from the tope implementation, thereby returning an `isa` score for each string. Based on these scores received back through the API, the add-in flags invalid strings with small red triangles.

To summarize, there are three key parts of the TDE that together support reuse. Working outward from the application, these parts are the following: add-ins to connect applications to the TDE via an API, a local copy of the repository software to support local search-by-match, and a remote repository server so that users can share tope implementations with one another.

The following sections describe these three parts in detail. Section 7.1.1 describes how the add-ins and TDE API make it possible for tope implementations to be application-agnostic. Section 7.1.2 describes the TDE's support for finding tope implementations on the local machine using the search-by-match algorithm. Section 7.1.3 broadens this discussion to include issues related to sharing tope implementations via a remote repository.

7.1.1 Supporting application-agnosticism of tope implementations

Topes are application-independent abstractions. For example, a string matches a conceptual abstraction of a phone number regardless of whether the string is in a spreadsheet or a web form. As far as the abstraction goes, the application does not matter.

Correspondingly, data descriptions in the TDE are application-independent, and the automatically generated tope implementations exist independently of whether their `isa` and `trf` functions are used on data from spreadsheets, web forms, or any other source. Architecturally, this represents a separation of concerns: the tope implementation contains functions for validating and reformatting data, with all application-specific concerns totally factored out of the tope implementation. As far as the abstraction implementation goes, the target application does not matter. This architecture helps to increase the reusability of tope implementations.

At some point, however, it is necessary to address application-specific concerns. For example, when validating a spreadsheet, code of some sort must read strings out of the spreadsheet cells and later act on `isa` functions' outputs by showing error messages for low `isa` scores, or updating the spreadsheet's user interface based on the output of `trf` functions.

Addressing these application-specific concerns is the role of an add-in. Each add-in essentially acts as a liaison between an application and the TDE. The add-in issues calls from the application to the TDE via an API provided by the TDE. To date, add-ins have been developed for several applications: the Robofox web macro tool, IBM's Co-Scripter web macro tool, the web form design tool in Microsoft Visual Studio.NET, and Microsoft Excel.

An example add-in connecting Robofox to the TDE

A web macro tool like Robofox [52] watches end-user programmers perform operations in a web browser and attempts to determine the intent behind those actions. The tool generates a macro to represent that intent (generally as a sequence of steps) which the tool can later execute on new data. For example, with Robofox watching, end-user programmers might go to a certain URL and copy a person's name from a particular tag in the web page. In order to identify which tag was selected, Robofox would record the tag's XPath and HTML ID attribute. In addition, it would record a visual pattern, which is an expression that identifies tags based on their proximity to key labels such as “Name:”. EUPs could demonstrate pasting the person’s name into a web form on another page. When Robofox replays the macro later, it would go back to the original URL, find the tag referenced by the XPath, ID and visual pattern, copy whatever text appeared in that location at runtime (which might be a different name than when Robofox created the macro), then paste the name into the form on the next page.

Unfortunately, web sites evolve: There have been many cases where webmasters changed pages’ structure, added or removed form fields, and made other changes that would have confused web macro tools [52]. When Robofox executes the macro described above, the person name tag might have moved on the page. While Robofox has sophisticated heuristics that combine XPath, ID and visual path information to locate moved tags, these heuristics might lead Robofox to locate the wrong tag—such as a tag containing a social security number, credit card number, or some other data. Robofox would then proceed to execute the macro using this wrong information. For example, this might cause Robofox to paste a social security number into the form in the other page, which obviously would be highly undesirable.

To counter errors of this kind, the developers of Robofox at the University of Nebraska have used the TDE to extend Robofox with assertions based on tope formats.[7] The add-in is a JavaScript library called by Robofox’s core application code to access the Toped-based TDE. When constructing a macro, end-user programmers can highlight a clipboard item, which is a variable that is initialized by a copy operation in the macro, and use the TDE to create a new format or select an existing format stored on the computer. Robofox then creates an assertion specifying that after the copy operation, the clipboard should contain a string that matches the specified 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 users to modify the macro or cancel execution if necessary. For example, if the user specified that the clipboard should match a person name, but at runtime the clipboard contains a social security number, then Robofox would display a warning popup before proceeding to send the social security number to another server. Thus, topes can detect errors that would otherwise have gone unnoticed.

[7] Available for download at http://esquared.unl.edu/wikka.php?wakka=RobofoxDownload (click on v0.2.3).

An example add-in connecting CoScripter to the TDE

IBM's CoScripter web macro tool—formerly called "Koala" [64]—has a new component called Vegemite [63], which allows end-user programmers to copy and paste data from web pages into tables in a "scratch space". While end-user programmers can type strings directly into scratch space cells, they can also create web macros that compute table cell values by posting other cell values through a web form and retrieving strings from the web server. Thus, CoScripter could automate many operations involved in user tasks uncovered by the Carnegie Mellon Contextual Inquiry (Section 2.3). For example, it could automate the process of gathering employee roster information from web sites into a spreadsheet. To date, however, CoScripter has lacked primitives for automating the reformatting operations involved in these tasks, such as reformatting a person name like "Chris Scaffidi" to "Scaffidi, Chris". Topes fill a critical need in automating these reformatting operations.

To support string reformatting, IBM used the Toped-based TDE API to implement a proof-of-concept feature as a popup menu in Vegemite. As with Robofox, the Vegemite add-in is a JavaScript library. After populating a cell value with a string, an end user can right-click on the cell value and select "Copy", which copies the string into the system clipboard. (Alternatively, end-user programmers could put a string on the system clipboard by executing a system copy command in another application.) Clicking on an empty cell and selecting "Paste Special" displays all possible reformatted versions of the string on the clipboard (Figure 30). End-user programmers can select a version, which Vegemite then pastes into the cell.

Vegemite populates the list of options by calling the TDE through the API to get a list of the available tope implementations, then iterating through them and calling the TDE via the API to test the string with each tope implementation. For each successful parse, Vegemite calls the tope's transformation functions (again through the API) to generate previews of the value as it would appear in the tope's other formats. In addition, Vegemite displays what parts of the string could be extracted from the value. Vegemite builds this list of parts by inspecting the data description (via a TDE API function) for the selected tope implementation.

For example, Figure 30 shows the options for pasting a phone number. Figure 31 shows a table whose second column was populated by copying each cell of the first column and pasting the state abbreviation (which was extracted through a tope that recognizes strings in “City, ST” format). The third column was populated by copying each cell of the second column and pasting the string using the state name format.

Figure 30. Copying and pasting a phone number using the add-in for CoScripter/Vegemite

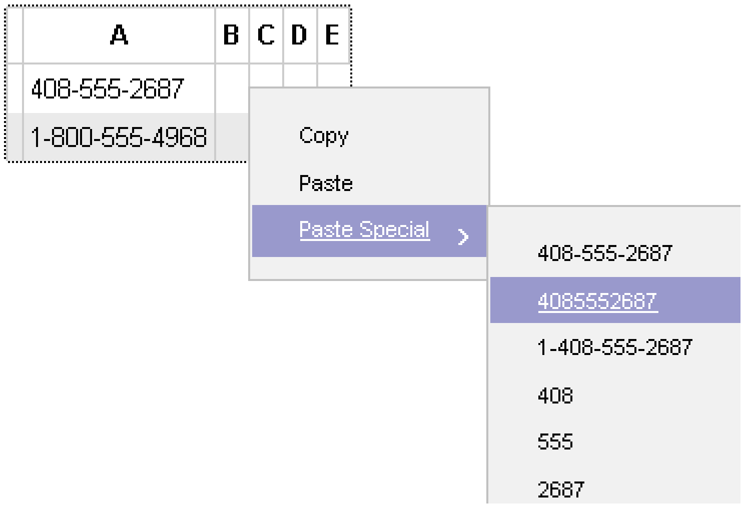
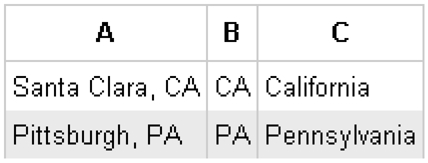


Figure 31. A Vegemite table containing two columns populated by calling tope transformations



A	B	C
Santa Clara, CA	CA	California
Pittsburgh, PA	PA	Pennsylvania

This proof-of-concept has three limitations to be addressed in future versions of Vegemite. First, Vegemite provides no user interface buttons or other controls to launch Toped, so end-user programmers presently have no way to add new formats via Vegemite. Second, this feature operates on individual cells; to save end-user programmers time, IBM could extend this feature to iteratively populate every cell in a column using the selected reformatting or extraction operation. Third, while the feature allows end-user programmers to populate cells through direct manipulation, selected operations are not recorded for replaying later on different data; addressing this limitation will require improved integration between Vegemite and the replay facilities of CoScripter. While these three limitations highlight further opportunities to improve the usability of Vegemite, they do not diminish topes’ contribution as a mechanism to recognize and reformat string data.

An example add-in connecting Microsoft Visual Studio.NET to the TDE

Even professional programmers often omit validation for input fields, including many form fields in "person locator" web applications that were created in the aftermath of Hurricane Katrina (Section 2.4). End-user programmers generally have less programming training than professional programmers, and tools for end-user programmers offer no more support for validation than do tools for professional programmers—in both cases requiring programmers to create a regular expression or a script to effect validation [11][95]. Thus, even more than professional programmers, end-user programmers probably struggle to implement validation with existing techniques.

As in web macros, the fundamental problem is that for many kinds of data, it is difficult to conclusively determine validity. For instance, no regular expression can definitively distinguish whether an input field has a valid person name. For any regular expression that does a reasonably good job of checking person names, there is almost certainly a valid person name that violates the regular expression. The "binary" validation of a regular expression is tolerable in the web macro and spreadsheet domains, where users can simply dismiss warning messages. In contrast, this limitation becomes a serious problem in the web application domain, since there is no way for web application users to override an "overzealous" regular expression that rejects an unusual but valid input. Consequently, web application designers often omit validation so as to avoid rejecting any valid inputs.

Topes offer a solution: warn the application user about questionable inputs, so that they can be double-checked rather than rejected outright, as demonstrated with an add-in for the web form design tool in Microsoft Visual Studio.NET (which comes in an Express Edition for end-user programmers).

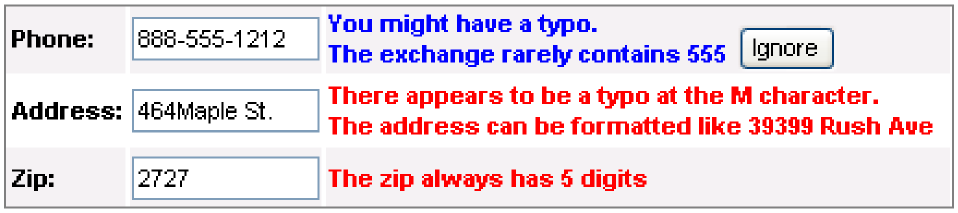
In Visual Studio.NET, the normal way for end-user programmers to implement validation code is to drag and drop a textbox widget from a toolbox onto the web form, then to drag and drop a `RegularExpressionValidator` widget alongside the textbox. They then specify a regular expression and a fixed textual error message. At runtime, if an input violates the regular expression, then the error message appears in red to the right of the textbox.

The Visual Studio.NET add-in is a new validator widget. After dragging and dropping a textbox, end-user programmers drag the new widget from the toolbox and drop it alongside the textbox. Once dropped on the page, the validator gives the option of selecting an existing tope, or creating a new tope by typing in examples of the data to be validated. The validator passes these examples to the Toped^{++}-based TDE via the API, which infers a format from the examples and presents it for review and customization before the tope's description file is stored on the web server by the add-in.

End-user programmers can add additional formats, if desired, and select a preferred format that this textbox's inputs should match. The validator automatically uses the tope to generate the necessary code for validating inputs. At runtime, this code calls the TDE API to reformat the input string, if necessary, to the preferred format. The code then calls the TDE API to validate the resulting string with the preferred format's `isa` function.

If the string matches the preferred format perfectly, then the code accepts the string. If the string does not match the format's grammar at all, or violates an "always" or "never" constraint, then the input is rejected; error messages are displayed using the standard red text beside the textbox. If the string is questionable, the generated code displays an overridable blue warning message so the end user can double-check and possibly correct the string before it is accepted. The application's programmer can also specify alternate settings (by configuring the property-value list of the validator in Microsoft Visual Studio), such as always rejecting any input that does not match the preferred format exactly, thus dispensing with overridability. Figure 32 illustrates the resulting web application user interface by depicting a web form for collecting information similar to the data that Hurricane Katrina developers wanted to collect.

Figure 32. Validation error messages alongside the input textboxes in a web form
In this example, the first error message is overridable, while the second and third are not.



Phone:	888-555-1212	You might have a typo. The exchange rarely contains 555 [Ignore]
Address:	464Maple St.	There appears to be a typo at the M character. The address can be formatted like 39399 Rush Ave
Zip:	2727	The zip always has 5 digits

An example add-in for Microsoft Excel

Section 4.1 already presented the Microsoft Excel add-in. This add-in appears as a toolbar in Microsoft Excel (Figure 9). The user can highlight a column of spreadsheet cells, then click the "New" button of the add-in. The add-in reads the highlighted cells and passes their strings into the Toped^{++}-based TDE (via the API), which infers a boilerplate data description and presents for customization and review by the user; alternatively, the user can right-click on a column of cells, which the add-in passes to the TDE to retrieve a list of recommended tope implementations for the user to choose from (Figure 15). After the user selects a tope implementation, the add-in can pass strings back into the TDE in order to call `isa` functions. The add-in creates small red triangles (Excel comments) to flag cells with low `isa` scores (Figure 12). While the current add-in does not visually distinguish between questionable and definitely invalid strings, it does provide a drop-down menu (on the toolbar) enabling users to filter out errors for questionable strings, to show only error messages for definitely invalid strings. If the user clicks the "Reformat" button to select a format in the tope implementation, then the add-in passes each cell's string into the TDE to call the `trf` functions, then updates the user interface with the resulting strings.

Implementing an add-in

Since they are application-independent, tope implementations can be reused without modification in other applications. Leaving aside nuances about whether specific add-ins are integrated with the Toped-based TDE or the newer Toped^{++}-based TDE, every one of the aforementioned add-ins can call exactly the same tope implementations. For example, a user could create a tope implementation for validating data on a spreadsheet, then reuse that implementation for validating web form inputs.

While the architecture insulates the tope implementation from tool-specific concerns, the architecture has a second advantage: it helps to *insulate the application* (Robofox, CoScripter, Microsoft Visual Studio.NET, or Excel) from concerns related to tope implementations. In other words, application developers can integrate topes into their applications without having to worry about the gory details of inferring data descriptions from examples, presenting data descriptions for editing, generating context-free grammars, and so forth. Application developers simply need to create an add-in that calls TDE API functions and updates the application's user interface. This API is available in C# or in Java and can be downloaded as open source. [8]

[8] http://www.topesite.com

Another benefit of wrapping tope implementations with the TDE API is that it simplifies add-in code required to call `isa` and `trf` functions. For example, the TDE API provides a function that can be called to compute the maximal `isa` score when a string is tested with all of a tope implementation's `isa` functions. This makes it possible to test, with a single line of code, whether a string matches *any* of the tope's formats. In fact, the TDE API returns more than just a raw `isa` score: it also returns the automatically generated human-readable description of any constraints violated by each invalid string, so that add-ins can display targeted error messages.

To concretely illustrate what is required to create an add-in, the following discussion delves into the implementation of the Excel add-in. For simplicity, the discussion below presents pseudo-code (Figure 33), rather than actual implementation code. Note that each event handler only needs to call one or two `Topes.API` methods. This helps to keep the handlers extremely short, even after exception-handling code is added to the pseudo-code.

The C# `Topes.API` class serves as a façade [36] for calling the TDE (Figure 34). Most add-ins simply call this class's methods. For example, when the user highlights cells and clicks the Excel add-in's "New" button, the add-in's event handler for this button passes the cells' text into `Topes.API.InferEditSave()`, which infers a data description from the examples, presents it for customization and review, saves it to the local computer, and returns the tope's globally unique identifier (GUID string) back to the add-in.[9] The add-in stores this GUID in an annotation attached to each highlighted cell. This annotation does not actually validate the cells; it just records the appropriate GUID for each cell. After recording the annotation, the add-in passes the cells' text and the tope GUID to `Topes.API.BulkValidate()`, which validates each cell's text and returns an `isa` score for each, along with an error message for invalid strings and a copy of the text that was passed in. The add-in takes each error message and attaches it as a "comment" tooltip to the corresponding cell.

[9] "GUIDs" are Microsoft's implementation of the universally unique identifier (UUID) standard specified in RFC 4122. UUIDs are pseudo-random 128-bit numbers usually represented as a hexadecimal string used to identify a computational artifact. UUIDs are well-established and widely used in a range of different systems, including Microsoft Windows and Intel hardware chips. Microsoft's guid-generation algorithm "never produces the same number twice, no matter how many times it is run or how many different machines it runs on" http://msdn.microsoft.com/en-us/library/ms221064.aspx.

The add-in also has an event handler (not shown in Figure 33) that is called when the user edits any cells. This handler revalidates the cells if they have an annotation specifying a tope GUID. In addition, the add-in has an event handler for the toolbar's drop-down widget (Figure 9), in order to hide or show tooltips for questionable strings. Finally, the add-in has a context menu (discussed further by Section 7.1.2) that allows the user to browse through existing tope implementations; the add-in populates this menu by calling `API.Search()`, which returns an enumeration of matching topes on the computer.

Figure 33. Pseudo-code for the primary functionality of the Excel add-in

In the "New" button's event handler…

```
topeGuid = Topes.API.InferEditSave(highlightedCells)
validationResults = Topes.API.BulkValidate(highlightedCells, topeGuid)
displayValidationResults(highlightedCells, validationResults, topeGuid)
```

In the "Load" button's event handler…

```
topeGuid = Topes.API.BrowseForTope()
validationResults = Topes.API.BulkValidate(highlightedCells, topeGuid)
displayValidationResults(highlightedCells, validationResults, topeGuid)
```

In the "Edit" button's event handler…

```
topeGuid = highlightedCells[0].annotation
topeGuid = Topes.API.EditSave(topeGuid)
validationResults = Topes.API.BulkValidate(highlightedCells, topeGuid)
displayValidationResults(highlightedCells, validationResults, topeGuid)
```

In the "Reformat" button's event handler…

```
topeGuid = highlightedCells[0].annotation
formatGuid = Topes.API.BrowseForFormat(topeGuid)
validationResults = Topes.API.BulkTransformAndValidate(
     highlightedCells, topeGuid, formatGuid)
displayValidationResults(highlightedCells, validationResults, topeGuid)
```

A helper function used by the event handlers above…

```
displayValidationResults(
        List<string> cells, List<IsaResult> isaResults, string topeGuid)
for (int i = 0; i < cells.Count; i++)
     cells[i].annotation = topeGuid
     cells[i].tooltip = isaResults[i].Message
     cells[i].text = isaResults[i].Text
```

Figure 34. Some key methods from the C# class Topes.API

```
// Shows a dialog window where the user can choose a data description, then
// returns a tope guid to the caller
public static string BrowseForTope()

// Presents a dialog so the user can choose one of the tope's formats, then
// returns a format guid to the caller
public static string BrowseForFormat(string topeGuid)

// Validates the strings and returns an IsaResult for each.
// Note that an IsaResult object (as returned by the TDE's Topep module)
// provides more than just an isa score.
// Each IsaResult contains an isa score and possibly a non-null error message.
// Each IsaResult also repeats back the string that was used to generate the IsaResult.
public static List<IsaResult> BulkValidate(
                                      List<string> strs, string topeGUID)

// Transforms each string to the selected format,
// validates each string against the format's isa function, and
// returns an array of IsaResults.
// Each IsaResult contains a non-null string indicating transformation output,
// as well as an isa score for this non-null output string and possibly a non-null error message
public static List<IsaResult> BulkTransformAndValidate(
           List<string> strs, string topeGUID, string targetFormatGUID)

// Presents a data description for editing and returns a tope guid
// (which may differ from the original topeGuid,
// if the user chooses to replace the tope with another tope)
public static string EditSave(string topeGuid)

// Infers a data description from the examples, presents it for review and customization, saves it,
// and returns a tope guid to the caller; pass null if no examples are available
public static string InferEditSave(IEnumerable<string> examples)

// Returns a list of all the topes known to the system.
public static IEnumerable<string> ListAllTopes()

// Finds tope implementations that matching a query
// (space-delimited keywords, and/or list of examples),
// returning the tope guid, name, and stars for each tope implementation
// (1-4 stars, depending on how well the tope matches the query)
public static IEnumerable<Tuple<string, string, float>> Search(
                   string keywords, IList<string> examples)
```

The amount of code required for interacting with the TDE API is extremely small compared to the code overall required to write the Excel add-in. Of the 1160 lines of code in the add-in, only 10 actually make calls to the TDE API. Another few dozen lines call methods exposed by objects returned by the API. The remaining 1000 lines of code relate to presenting security credentials so that Excel will load the add-in, initializing the add-in's toolbar, registering the nested context menu, reading/writing values from/to the spreadsheet cells, exception handling, and displaying an "About this Add-In" informational screen. In short, the amount of code required to interact with the TDE API is far smaller than the amount of code required to interact with Excel. Once a programmer has mastered the process of writing any add-in for Excel, there is relatively much less that must be learned in order to interface to the TDE API.

Steps for implementing an add-in

The essential steps for implementing an add-in for a new application are the following:

1. Decide which user interface widgets to validate/reformat with topes. Topes operate on strings. So fire up the application and find the fields that have text in them. In a web form, the text is in text fields. In a spreadsheet, the text is in cells. The key question to answer is, "Where do the strings live in the application?"
2. Add controls to the application's user interface so users can select a tope for the text widgets. For example, the Excel add-in provides a button for this (in the toolbar) as well as a context menu, while the CoScripter/Vegemite add-in provides a context menu. Choose whatever control fits in most seamlessly with the style of user interaction supported by the application (e.g.: if the application generally provides context menus for interaction, then use a context menu rather than a drop-down widget or button). In each control's event handler, call the TDE's API.
3. At runtime, pass the text widget's value, a string, to the TDE API for validation and reformatting. Receive the API's return values and update application user interface. Again, follow the standard user interaction style for displaying outputs. Web forms generally display error messages to the right of text boxes, so that is the approach taken by the TDE add-in for Microsoft Visual Studio.NET. Excel usually uses modeless flags on cells (usually little green triangles), so that TDE add-in takes a similar approach.
4. Try out the resulting add-in for a while, perhaps in a user study, and iterate as needed. My first add-ins for Excel and Visual Studio were fairly unattractive and slow because I didn't really know the applications' legacy interfaces too well. My second attempts were better, as I learned out how to select more attractive icons, how to read/write data from the application more efficiently, and how to add context menus to Excel. The most recent version of the add-in is very good, with ex-

tremely attractive graphics that took hours to make, amazingly fast read/write from the application, and a context menu that users found extremely intuitive to use (Section 6.2.2).

7.1.2 Recommending tope implementations that match user-specified examples

As described by Section 5.2.2, 32 topes accounted for 70% of all spreadsheet columns that could be categorized in a corpus of spreadsheets from the web. 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, the Excel add-in shows a list of recommended topes in a context menu when the user right-clicks on a group of highlighted cells (Figure 15). The add-in retrieves this list via a call to `API.Search()`, a function that accepts two parameters: a list of keywords and a list of example strings. The add-in gets the keywords from the first row above the highlighted cells (since a column's first cell often has a header describing the column [1]) and gets the example strings from the highlighted cells.

This context-menu is actually functionally redundant with the dialog window displayed in a call to `API.BrowseForTope()`, since this dialog window also allows users to enter keywords and examples to search for tope implementations. Displaying the list in a context-menu simply provides a tighter integration with the user interface of the Excel application. As described by Section 7.1.3, the repository also provides a similar search feature so that users can look for tope implementations on the server.

Basic search-by-match tope recommendation algorithm

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 35 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 w[τ] 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 by a score c[τ] indicating on how well each tope matches the examples. Because $0 \leq$ c[t] $\leq$ 1/(|E| + 2), c[τ] scores effectively break ties between topes with the same keyword and whitelist counts.

Figure 35. Pseudo-code for basic recommendation algorithm

```
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 $s_0$ such that
     at least N topes in T have s[t] ≥ $s_0$
 For each tope t in T such that s[t] < $s_0$
   Remove t from T
```

Optimizations and CharSigs

The pseudo-code above is straightforward to understand, but it is not quite fast enough for interactive use, so I 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. The same approach is used for whitelists.

Inverting the third loop to retrieve topes that match a certain example is more complicated. In contrast to retrieving tope implementations by keyword, the potentially infinite number of values matched by each tope implementation makes it impossible to index tope implementations according to what examples they match. Instead, tope implementations are indexed 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" with 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.

A new data structure called a CharSig records 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 from the tope's data description. 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. In a Hierarchical part, 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 have 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

$\mathtt{S}^{\mathtt{i}}_{\mathtt{cc}}$ of topes that could have `i` instances of character class `cc`. It also records $\mathtt{S}^{\infty}_{\mathtt{cc}}$, 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, a tope could only match the license plate number "ABC-DE20" if its instances could have 5 letters, and if its instances could have 2 digits, and if its instances could have 7 alphanumerics, and if its instances could 1 hyphen. The set S^{5}_{letter} contains the topes whose instances could have 5 letters, S^{2}_{digit} contains the topes whose instances could have 2 digits, S^{7}_{alnum} contains the topes whose instances could have 7 alphanumerics, and S^{1}_{-} contains the topes whose instances could have 1 hyphen. Thus, to find topes that could match "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 $\mathtt{O(|T|)}$, as shown by examining the steps of the algorithm. Computing keyword and whitelist hits with an inverted index is trivially $\mathtt{O(1)}$ for each string presented by the user. Each set $\mathtt{S}^{\mathtt{i}}_{\mathtt{cc}}$ has worst-case size of $\mathtt{O(|T|)}$, so intersecting those sets to get the list of candidate topes is $\mathtt{O(|T|)}$. Testing each tope in this set of $\mathtt{O(|T|)}$ topes is $\mathtt{O(1)}$, so the total cost of calling isa functions is also $\mathtt{O(|T|)}$. Finally, the intervening `Prune` operations use the $\mathtt{O(|T|)}$ 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).

While the simulation discussed by Section 7.2.2 empirically confirms that these optimizations produce an $\mathtt{O(|T|)}$ implementation, it also shows that the optimizations do help to improve performance by a constant factor. Specifically, the simulation 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 $\mathtt{S}^{\mathtt{i}}_{\mathtt{cc}}$ sets can be done in $\mathtt{O(1)}$ time with respect to the total index size.

In passing, note that a CharSig could also be computed bottom-up for regular expressions. Specifically, for the empty regular expression ε, every character class must occur 0 times. The character content of a single character `c` is 1 of the class for `c`. When a regular expression r_1 is concatenated with another regular expression `r1`, the lower (upper) bound of each character class in r_1r_2 is equal to the sum of the lower (upper) bounds in r_1 and r_2. In a disjunction of r_1 and r_2, the lower (upper) bound of each character class in $r_1|r_2$ equals the minimum (maximum) of the lower (upper) bounds in r_1 and r_2. When a regular expression `r` is wrapped with a Kleene star, the lower (upper) bound of each character class in `r*` is 0 (∞). This direct applicability of CharSigs to regular expressions suggests that it may be possible to use the tope recommendation algorithm to search regular expressions by example in future systems.

7.1.3 Support for publishing, finding, and combining tope implementations

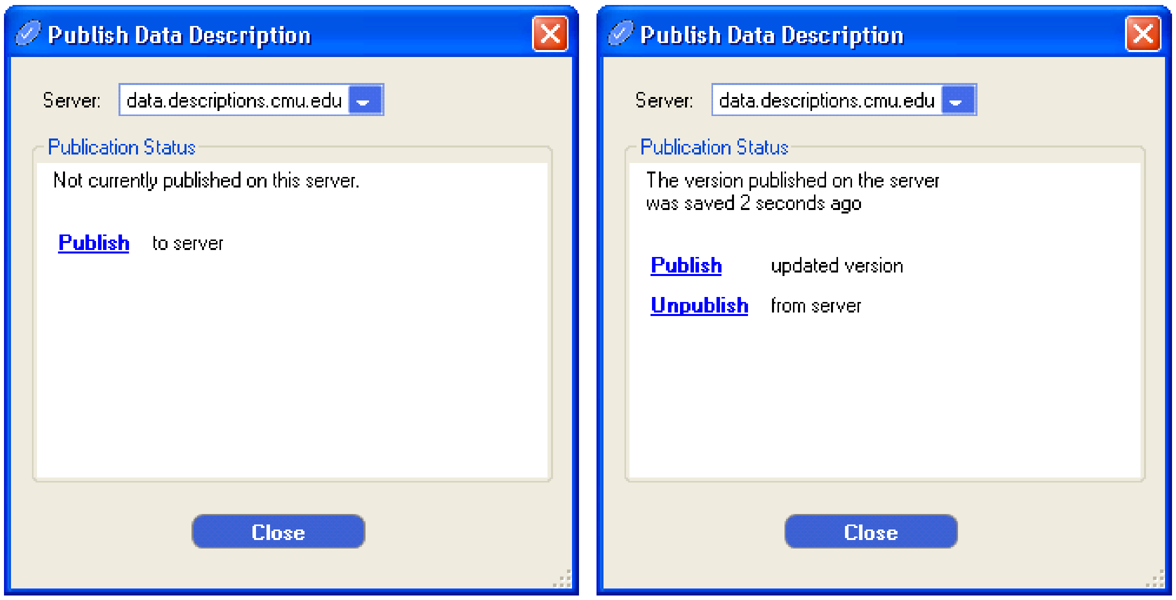
While the basic search-by-match algorithm provides a way for the TDE to recommend tope implementations that already exist on the user's computer, it is insufficient to support reuse of tope implementations by multiple people. The TDE's repository server and related user interfaces fill this need. First, the TDE provides a simple user interface for publishing tope implementations to a repository server. Second, the repository includes a web server application for searching a repository for tope implementations; Toped^{++} embeds an instance of Internet Explorer so that the user can access the repository via this web application without ever leaving the context of the data description editor. The repository's search interface presently uses the same search-by-match algorithm used for searching the user's local machine, though this algorithm could be parallelized in future versions of the repository.

Publishing a tope implementation

While editing a data description in Toped^{++}, the user can open the "Publish / Unpublish" dialog (Figure 36). Toped^{++} populates the list of servers by looking in a specific directory, where a repository descriptor file is stored for each server. This repository descriptor file, formatted in XML, specifies a URL for publishing and unpublishing data descriptions using the standard REST protocol [34] (where an HTTP POST or PUT operation publishes a new object, DELETE unpublishes it, and GET retrieves the latest copy). Whereas the REST URL is for programmatically reading and writing descriptor files in XML via HTTP, the repository descriptor file also specifies a “human interface” URL (discussed below) that a regular web browser can use to access the web site.

The user can add new repositories to the list of recognized repositories. This is accomplished in three steps. First, the repository’s owner puts a copy of the repository descriptor file on some web server somewhere and puts a hyperlink to it on a web page. Second, when a user who has the TDE installed clicks on that hyperlink, the TDE is alerted by Windows (through a special “asynchronous pluggable protocol event handler” that the TDE registers with Windows), and the TDE prompts the user to confirm that he wants to add the repository into the user’s list of recognized repositories (Figure 37). If the user confirms that this repository should be recognized, then the TDE stores the repository’s descriptor file on the local computer, and the repository is added to the user’s list of recognized repositories. The TDE does not presently have any dialog window for removing repositories from the list of recognized repositories, but this can be accomplished by simply deleting the repository’s configuration file from the hard drive. Future versions of the TDE will include a user interface for deleting configuration files so that the user can prune the list of recognized repositories, or for changing the directory where configuration files are stored.

Figure 36. Window for publishing / unpublishing data Toped^{++} descriptions (before and after clicking “Publish”)



Server: data.descriptions.cmu.edu

Not currently published on this server.

Publish to server

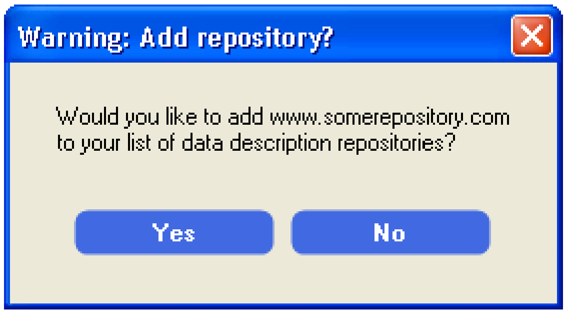
Server: data.descriptions.cmu.edu

The version published on the server was saved 2 seconds ago

Publish updated version

Unpublish from server

Figure 37. Adding a new repository to a user's list of recognized repositories


Warning: Add repository?

Would you like to add www.somerepository.com
to your list of data description repositories?

Returning again to the Publish/Unpublish dialog window (Figure 36), the "Publish" link allows the user to send a copy of the data description that he is currently editing up to the repository. The TDE accomplishes this by performing an HTTP POST operation to the REST URL specified in the repository's descriptor file. It immediately performs a GET to verify that the data descriptor was uploaded correctly. The "Unpublish" link allows the user to remove the data description from the repository. The TDE accomplishes this by performing an HTTP DELETE operation using the same URL.

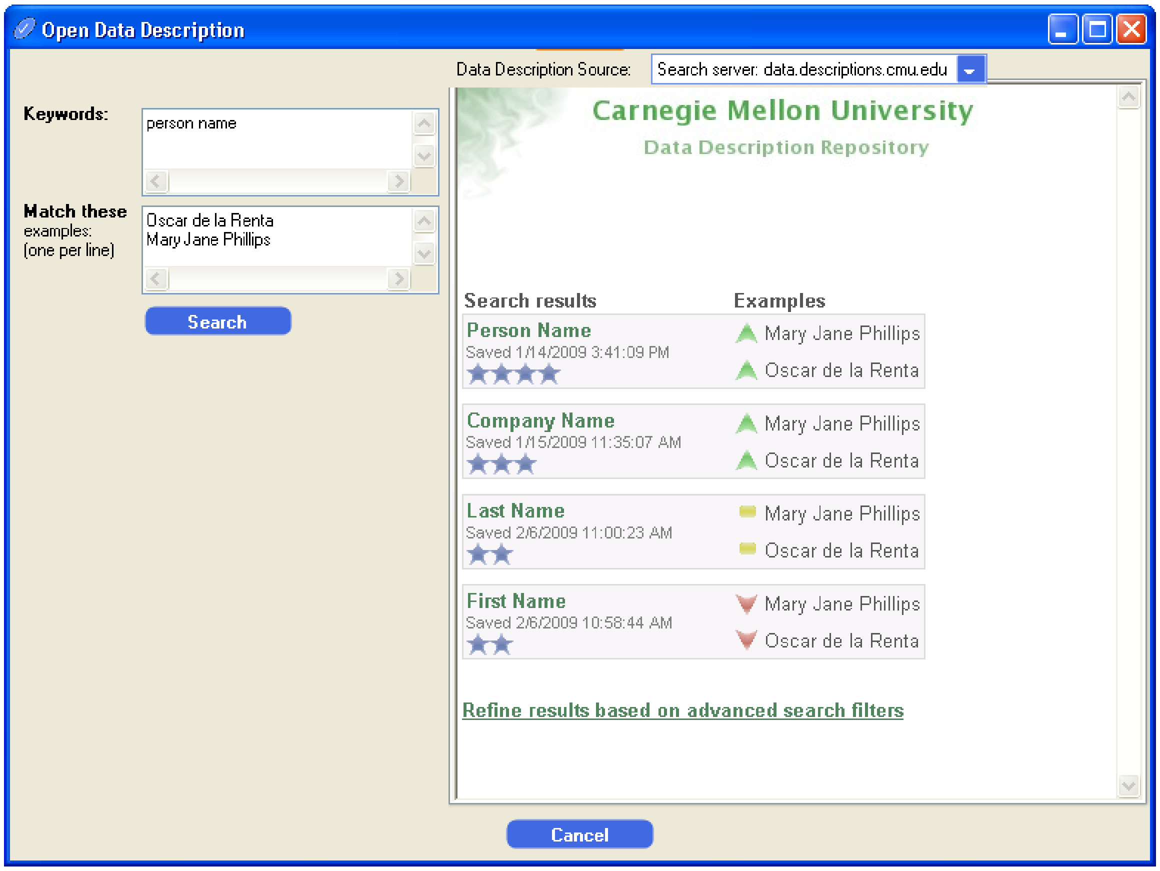
The current repository implementation is a simple wiki-style content management system. Any user can edit or delete any other user's content. The wiki does, however, require all users to have an account; so the first time that a user accesses a repository, he will be prompted to create an account by entering a new user name and password. Future versions of the repository may provide more sophisticated content management.

Finding and downloading tope implementations

The repository server includes a web application allowing users to search through the data descriptions currently stored on the server. This search interface is based on the same search-by-match algorithm supported on the user's local machine. The user can access it through a dialog window inside of Toped^{++} (Figure 38) or can access the web application through a browser (Figure 39). Toped^{++} achieves this by hosting an instance of Internet Explorer inside the dialog window and directing the browser to the "human interface" URL specified in the repository descriptor file.

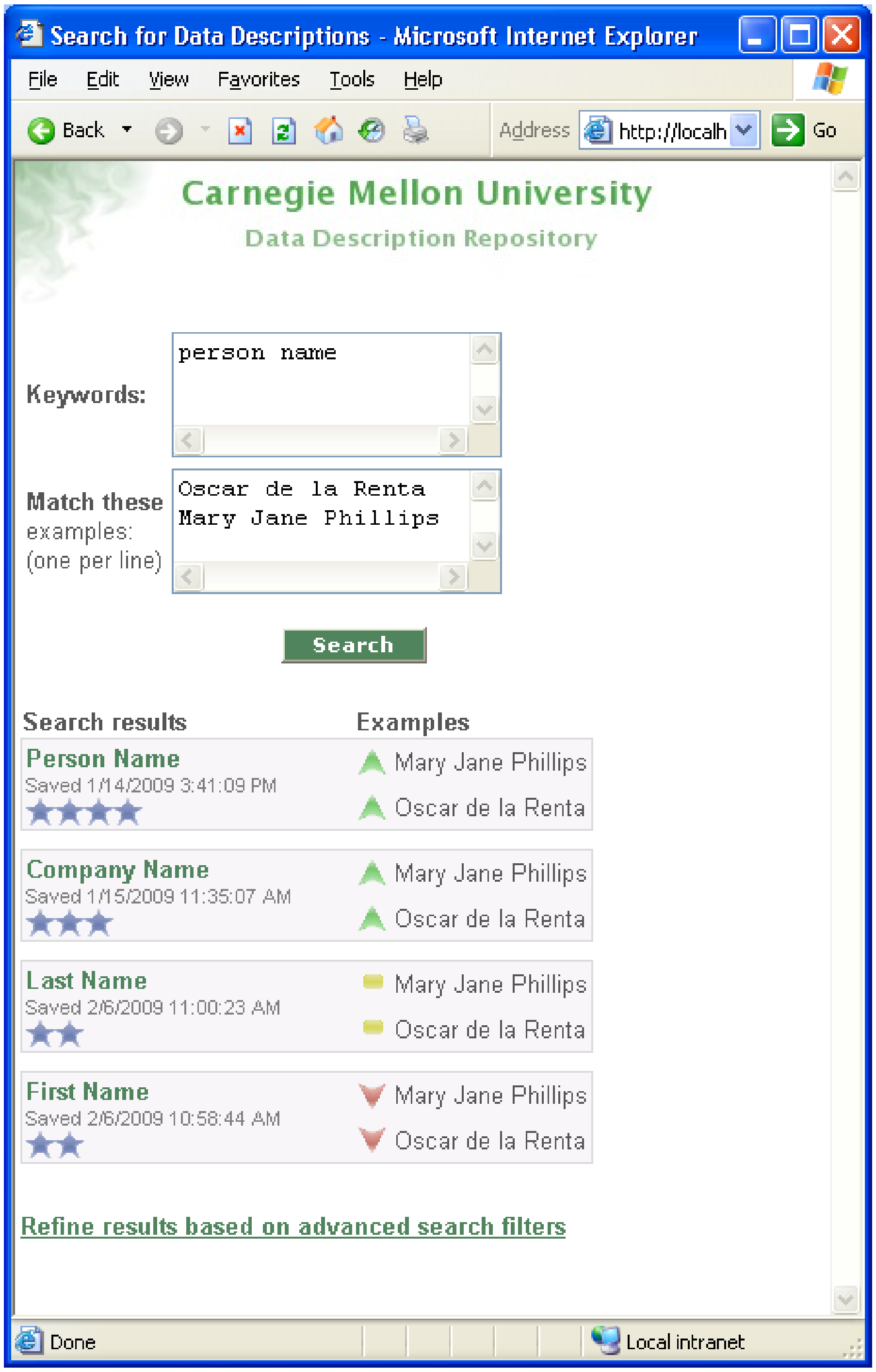
In either case, the repository uses icons to show how well each example string matches each tope implementation in the search (with green up arrows flagging good matches, yellow squares flagging questionable matches, and red down arrows flagging poor matches). One to four blue stars summarize the closeness of the each search result to the user's query—in the example shown in Figure 39, the "name" keyword hit the "Company Name", "Last Name" and "First Name" data descriptions, but these have less than four stars because they missed the "person" keyword, and in the case of the latter two search results, they even missed some example strings. The number of stars is approximately proportional to the score of the search result as computed in the search-by-match algorithm (the final value of `s[τ]` in Figure 35), where four stars corresponds to the maximal possible score possible by hitting all keywords and all example strings. (This proportionality is not precise because of rounding to the nearest number of stars.)

Figure 38. Searching the repository from within Toped++


Open Data Description
Data Description Source:
Search server: data.descriptions.cmu.edu
Keywords:
person name
Match these examples: (one per line)
Oscar de la Renta
Mary Jane Phillips
Search
Carnegie Mellon University
Data Description Repository
Refine results based on advanced search filters
Cancel

Search results	Examples
Person Name Saved 1/14/2009 3:41:09 PM ★★★★	▲ Mary Jane Phillips ▲ Oscar de la Renta
Company Name Saved 1/15/2009 11:35:07 AM ★★★	▲ Mary Jane Phillips ▲ Oscar de la Renta
Last Name Saved 2/6/2009 11:00:23 AM ★★	▬ Mary Jane Phillips ▬ Oscar de la Renta
First Name Saved 2/6/2009 10:58:44 AM ★★	▼ Mary Jane Phillips ▼ Oscar de la Renta

Figure 39. Using a web browser to search the repository


Search for Data Descriptions - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address http://localh
Go

Carnegie Mellon University

Data Description Repository

Keywords: person name

Match these examples: (one per line)

Oscar de la Renta
Mary Jane Phillips

Search

Search results	Examples
Person Name Saved 1/14/2009 3:41:09 PM ★★★★	▲ Mary Jane Phillips ▲ Oscar de la Renta
Company Name Saved 1/15/2009 11:35:07 AM ★★★	▲ Mary Jane Phillips ▲ Oscar de la Renta
Last Name Saved 2/6/2009 11:00:23 AM ★★	▬ Mary Jane Phillips ▬ Oscar de la Renta
First Name Saved 2/6/2009 10:58:44 AM ★★	▼ Mary Jane Phillips ▼ Oscar de la Renta

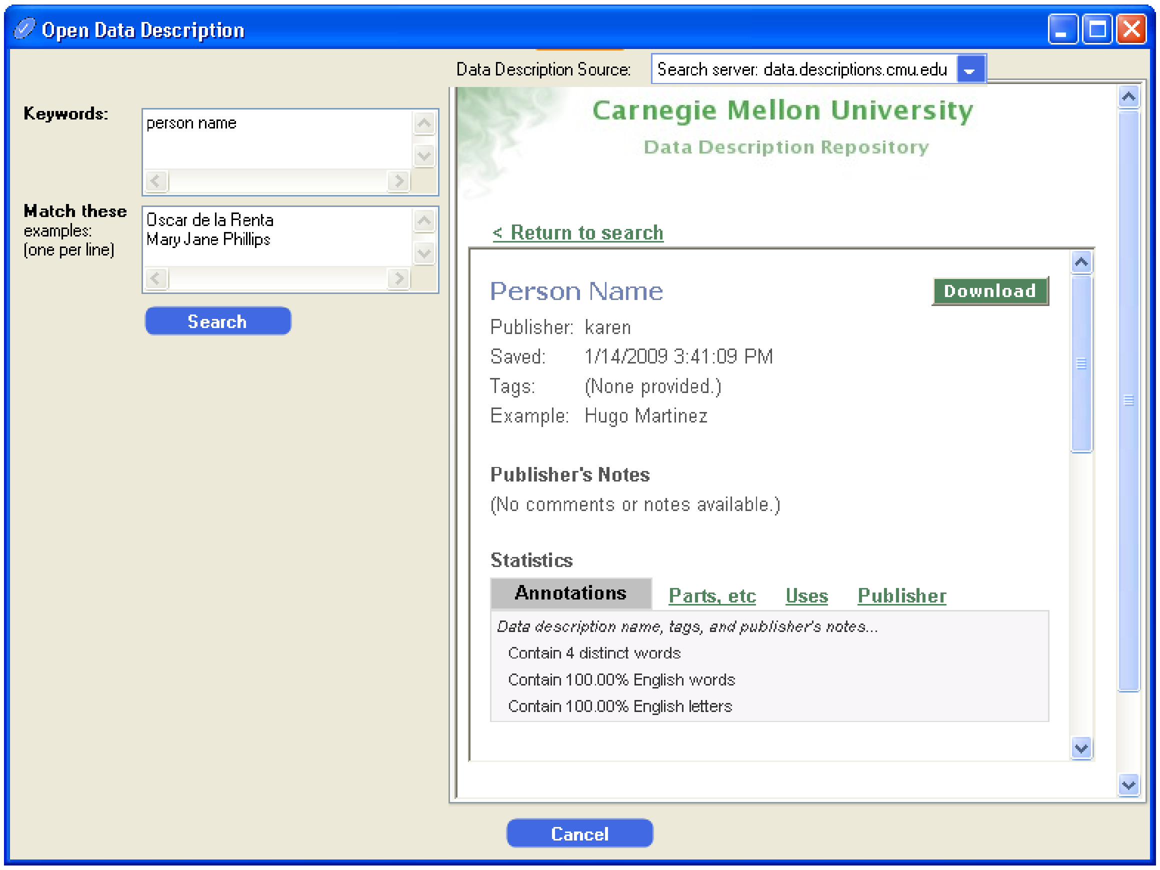
Refine results based on advanced search filters

The search results page also shows a link allowing the user to engage in an "advanced search", which Chapter 8 describes in detail. This advanced search provides features for filtering search results based on criteria related to the apparent reusability of each tope implementation.

If the user clicks on one of the search results, the web application displays detailed information about the selected tope implementation (Figure 40). This page shows the tope implementation's name, author, and various statistics. Scrolling down in the page shows comments by other users. These statistics and comments are used for filtering in the advanced search feature and are described in detail by the next chapter.

When the user views the web application through Toped^{++} (as in Figure 40) rather than through a web browser, a "Download" button appears. If the user clicks this button, then Toped^{++} is notified through a special event handler from the web page that it hosts, and Toped^{++} retrieves the data description's XML via the REST URL specified in the repository descriptor file. It saves this data description on the local computer, just as if the user had created the data description himself.

Figure 40. Reviewing detailed information about a tope implementation


Open Data Description
Data Description Source:
Search server: data.descriptions.cmu.edu
Keywords:
Match these examples: (one per line)
Search
Carnegie Mellon University
Data Description Repository
< Return to search
Download
Annotations
Parts, etc
Uses
Publisher
Cancel

Keywords: person name

Match these examples: (one per line)
Oscar de la Renta
Mary Jane Phillips

Person Name

Publisher: karen
Saved: 1/14/2009 3:41:09 PM
Tags: (None provided.)
Example: Hugo Martinez

Publisher's Notes
(No comments or notes available.)

Statistics

Data description name, tags, and publisher's notes...
Contain 4 distinct words
Contain 100.00% English words
Contain 100.00% English letters

Combining data descriptions

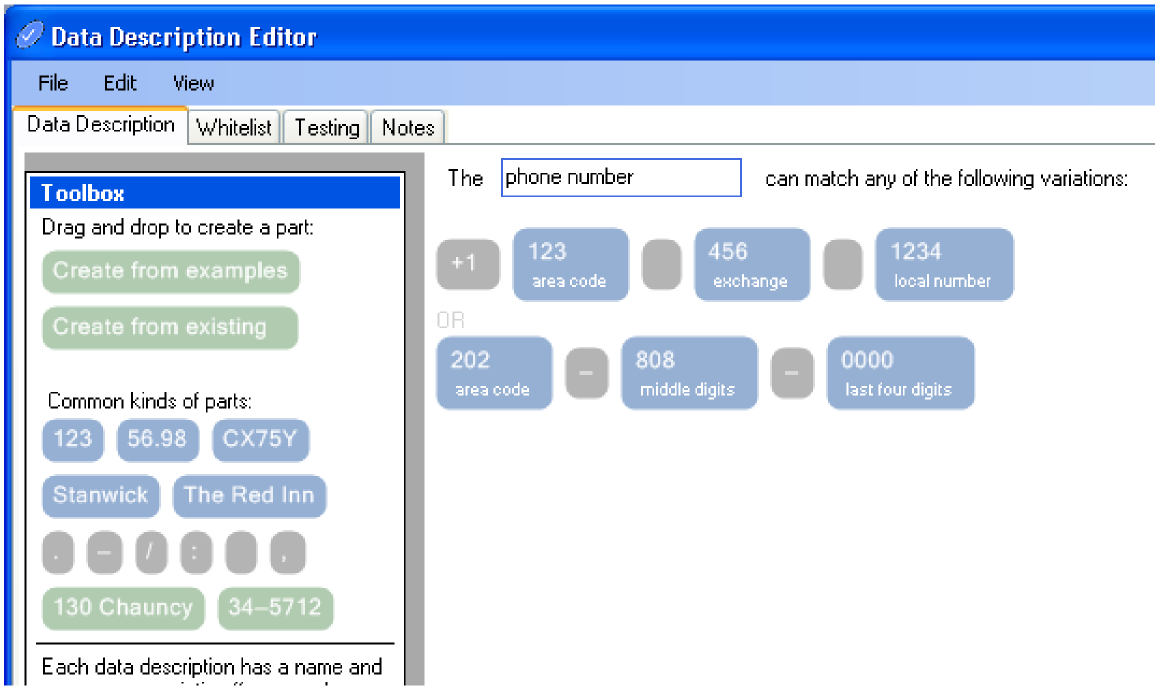
Once the user has stored a data description on the local computer, it can be used in exactly the same way as any other data description on the computer, regardless of where these descriptions came from. In particular, the user could combine data descriptions from multiple repositories. For example, the user could download a person name data description from one repository and an email address data description from another repository, then combine these to produce a data description for strings like "Christopher Scaffidi <cscaffid@cs.cmu.edu>".

Some caution is required, however, when the user wants to combine multiple data descriptions that each describe the *same* data category, since data descriptions might have different (inconsistent) names or specifications for parts. This caution is required regardless of whether the descriptions come from the same repository, but there probably is a higher likelihood of inconsistency when the descriptions come from different repositories, since different user communities might have different topes for the same kind of data.

For example, suppose that the user wants to combine a variation from one phone number data description with a variation from another phone number data description to produce a third data description. The act of combining the two variations is very simple. The user could open the first source data description in Toped^{++}, highlight the desired description, type Control+C to copy it into the clipboard, open the target data description, and hit Control+V to paste the variation. He could then repeat the same steps to copy a variation from the other source data description into the target data description.

After the user performs the steps described above, he will have successfully produced a data description containing two variations. Figure 41 illustrates what this data description might look like to the user. However, lurking beneath the surface, this resulting data description could have two kinds of inconsistencies.

Figure 41. Data description created by copying/pasting variations from other data descriptions



First, even though both variations have parts that play the role of an exchange and a local number, they do not use the same names for those parts. This will prevent the tope implementation generated by Topeg (Section 6.1.1) from successfully reformatting strings between formats in these variations. For example, if a string "+1 123 456 1234" is parsed with the first variation's grammar, then the parse tree will have a node labeled "area code" containing "123", a node labeled "exchange" containing "456", and a node labeled "local number" containing "1234". When the `trf` function targeting a format in the other variation executes, it will attempt to find parts called "area code", "middle digits", and "last four digits" (as in the algorithm shown in Figure 26). Because it will be unable to find the latter two parts, it will be unable to concatenate their values to the result string, so the algorithm will end up generating "123--". To catch errors of this kind at runtime, all outputs are tested against the target format's grammar and flagged for the user's review (Section 6.1.1). To help detect the *potential* for errors of this kind at *design time*, Toped^{++} presents a summary on the "Testing" tab of all parts that appear in only a subset of the variations (Figure 42).

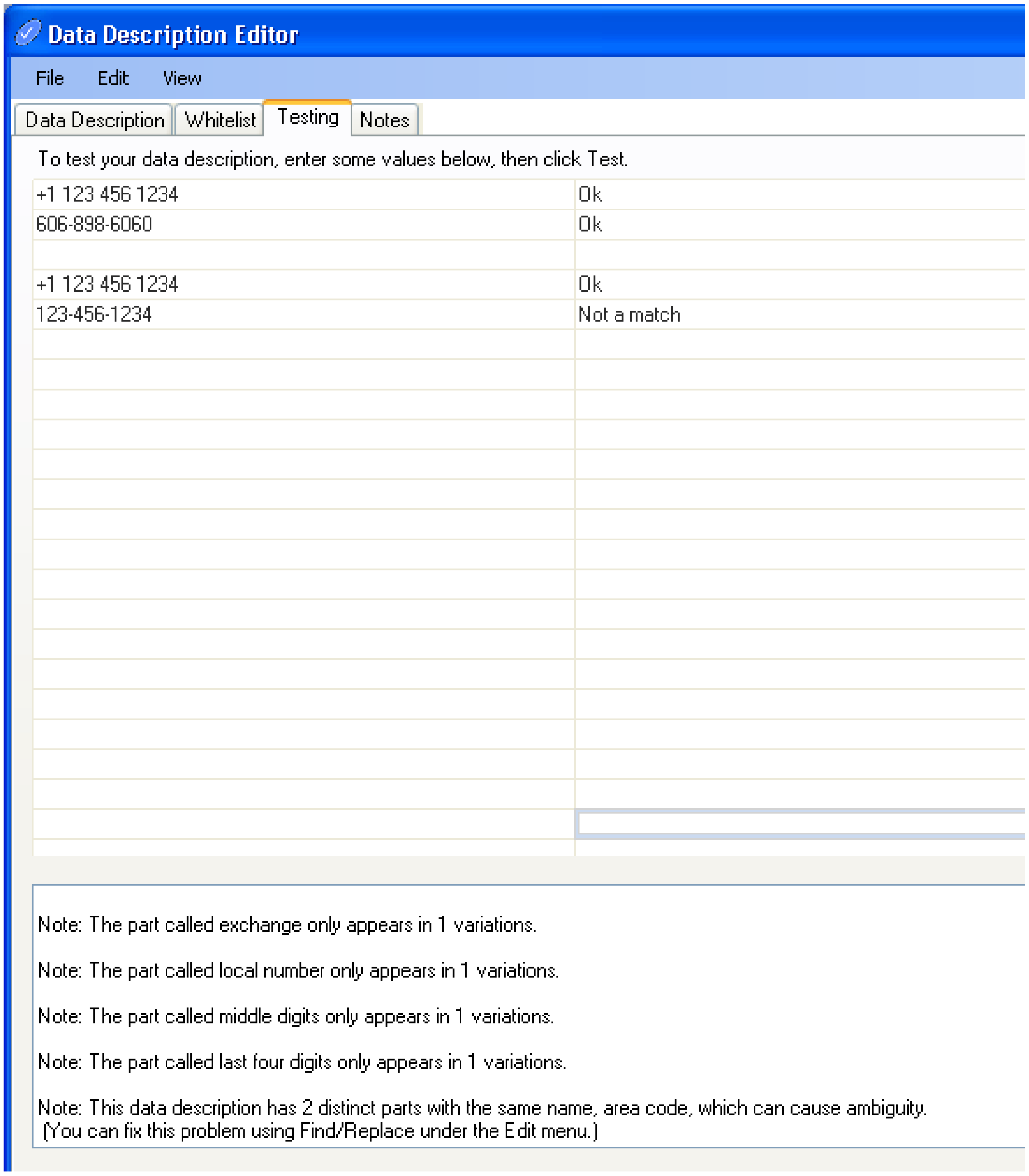
Second, even though both variations have a part named "area code", these two parts have a different definition. In this example, the area code part in the first variation does not have a constraint indicating that the area code needs to be 200 or greater, while the area code part in the second variation does have this constraint. So, for instance, the string "+1 123 456 1234" would match the first variation, but "123-456-1234" would not match the second variation. This inconsistency is not readily apparent in the user interface because the user must click on each part in order to view its constraints (though there is a clue to the presence of this inconsistency in that mousing over one of the two part icons will only highlight that part icon, rather than both part icons, which would happen if both icons corresponded to the same part specification). To help the user detect errors of this kind, the bottom half of the "Testing" tab of Toped^{++} displays a summary of parts that have multiple specifications (Figure 42).

To help the user fix both kinds of inconsistency, Toped^{++} provides a "Find and Replace" dialog window (Figure 43). For example, the user can replace all instances of the "middle digits" specification with the "exchange" specification, thereby addressing the first kind of inconsistency described above. The second kind of inconsistency is more subtle, as the user might need to actually view the parts' constraints in order to decide which specification to keep and which to replace. When making this decision, the user can click on a preview icon (👁) to examine a part specification's constraints. After replacing all instances of parts in the first variation with the corresponding parts from the second variation, the user will have remedied every inconsistency described above.

An alternate approach would be for Toped^{++} to aggressively detect the same kinds of inconsistencies at the moment that the inconsistencies are created (i.e.: when the user pastes the second variation into the data description), to interrupt the user and force him to immediately focus his attention on the inconsistencies, and to prevent him from moving forward with any additional editing activities until the inconsistencies had been addressed. This would prevent any of these inconsistencies from forming. On the other hand, it would probably be very annoying to the user (as modal interruptions are well-known to be distracting and frustrating), not to mention inconsistent with the style of the TDE overall (which identifies questionable behavior or data, then lets the user work at his own pace to double-check whether the problems are real).

In short, Toped^{++} does not prevent users from combining variations in an inconsistent manner. However, it provides testing features so that the inconsistencies are more obvious, as well as a simple dialog window for fixing these inconsistencies.

Figure 42. Detecting inter-variation inconsistency in a data description



To test your data description, enter some values below, then click Test.

+1 123 456 1234	Ok
606-898-6060	Ok
+1 123 456 1234	Ok
123-456-1234	Not a match

Note: The part called exchange only appears in 1 variations.

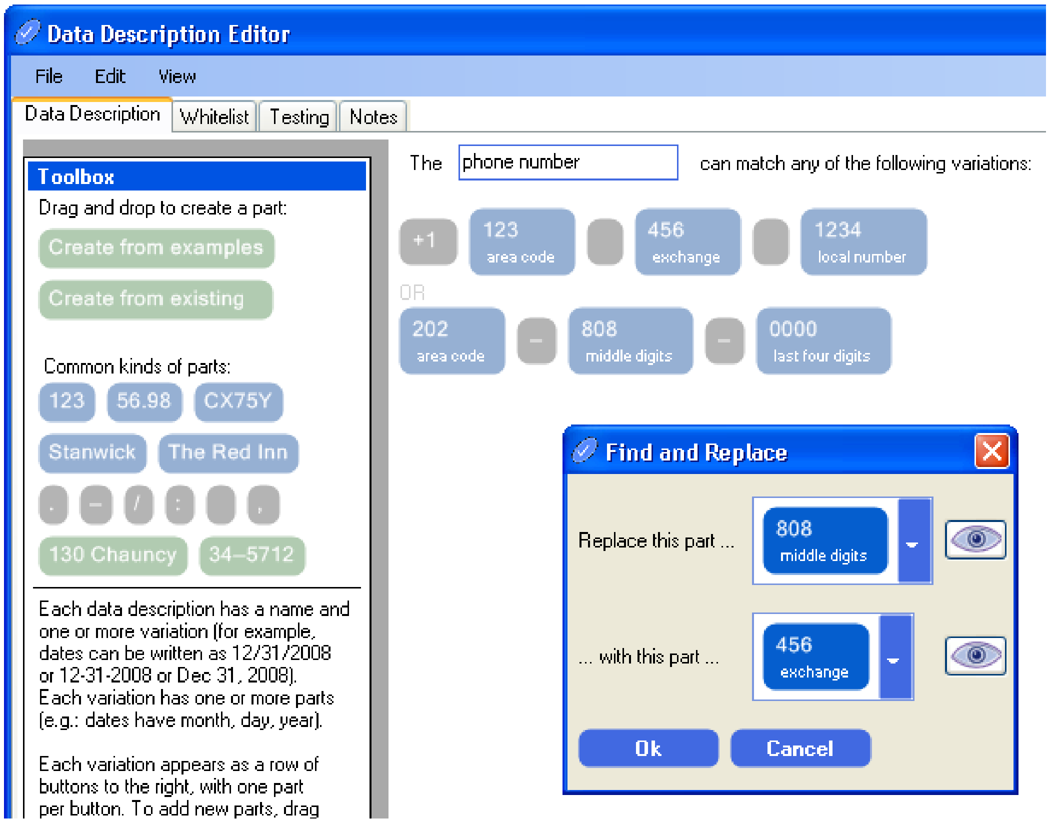
Note: The part called local number only appears in 1 variations.

Note: The part called middle digits only appears in 1 variations.

Note: The part called last four digits only appears in 1 variations.

Note: This data description has 2 distinct parts with the same name, area code, which can cause ambiguity.
(You can fix this problem using Find/Replace under the Edit menu.)

Figure 43. Using Toped^{++} to replace one part specification with another.

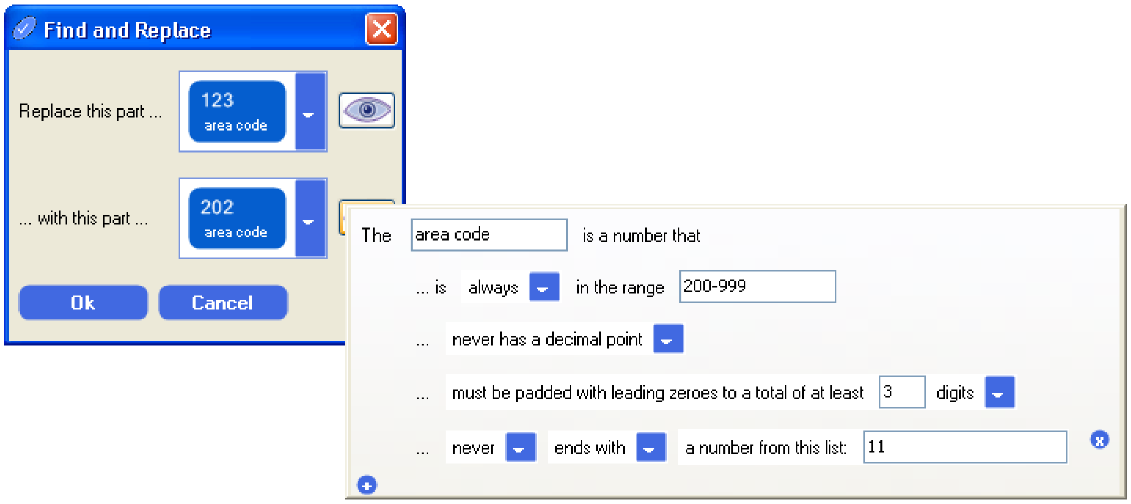


Each data description has a name and one or more variation (for example, dates can be written as 12/31/2008 or 12-31-2008 or Dec 31, 2008). Each variation has one or more parts (e.g.: dates have month, day, year).

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

The phone number can match any of the following variations:

Figure 44. Previewing a part's constraints in the Find and Replace dialog window of Toped^{++}



The area code is a number that

... is always in the range 200-999

... never has a decimal point

... must be padded with leading zeroes to a total of at least 3 digits

... never ends with a number from this list: 11

7.2 Evaluation

The reusability of tope implementations is predicated on a simple strategy: factor out all tope-specific concerns into an *application-agnostic* tope implementation that can be stored on a users' computer or a server, then *recommend* particular tope implementations based on keywords and examples of the data to be validated and reformatted. Therefore, the most crucial evaluation criteria for reusability are whether tope implementations really are application-agnostic and whether the search-by-match algorithm actually can recommend appropriate tope implementations in a reasonable amount of time based on just a few keywords and examples.

Section 7.2.1 addresses the first evaluation criterion with a simulation showing that tope implementations do indeed retain their accuracy when carried from one kind of application to another. Section 7.2.2 addresses the second evaluation criterion with a simulation showing that the search-by-match algorithm is fast enough and accurate enough for searching the user's computer. A qualitative analysis suggests that the algorithm offers straightforward opportunities for high-performance parallelization in a repository setting.

7.2.1 Accuracy of tope implementations when reused across applications

To evaluate reusability, I tested if the Toped-based topes that I implemented for spreadsheet data (Section 5.2.2) could accurately be reused to validate data from web applications. I used two applications as data sources for this experiment.

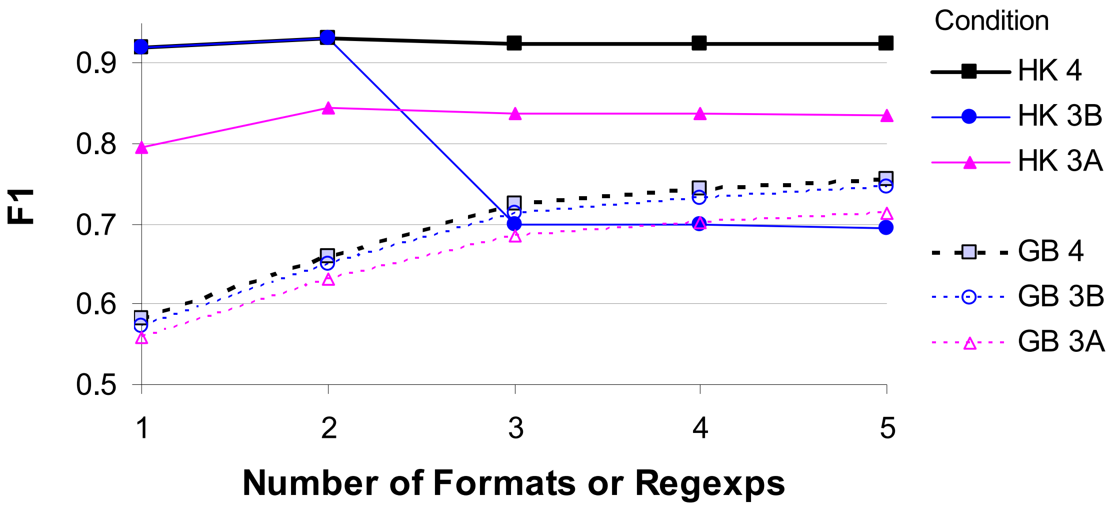
The first web application was Google Base, a site developed by professional programmers. Of the 32 spreadsheet-based tope implementations described by Section 5.2.2, 7 corresponded to fields in Google Base. I validated an eighth category in Google Base, mailing addresses, by concatenating existing address line, city, state, and country topes with separators and a zip code field. (That is, these topes corresponded to distinct cells in spreadsheets but were concatenated into one field in Google Base.)

The second web application was a Hurricane Katrina person locator site (described by Section 2.4), which was built by a team of professional and end-user programmers. Of the 32 spreadsheet-based tope implementations described by Section 5.2.2, 5 corresponded to text fields in this web application.

I manually validated 100 randomly-selected test values from each of the 13 categories and tested whether the tope implementations produced the correct validation scores. As in Section 5.2.2, I measured accuracy by computing the topes' F_1 for three different configurations (3A being the case where questionable strings were rejected, 3B being the case where questionable strings were accepted, and 4 being the case where a user would be prompted to double-check questionable strings). Comparing the results shown by Figure 45 to the earlier spreadsheet-based results shown by Figure 25 in Section 5.2.2, it is clear that the topes actually were *more* accurate when reused on the web application data than they originally were on the spreadsheet data. Specifically, the *highest* F_1 score on the spreadsheet data was approximately 0.7 (at the low end of acceptable), while here on web application data, the *lowest* F_1 score is 0.7 (and that is for cases HK 3B and GB 3A, which are cases that do not make use of topes' ability to identify questionable data for double-checking by users).

The main reason for this higher accuracy during reuse on web application data is that the web application data demonstrated a narrower range of different formats compared to the spreadsheet data. In fact, for Hurricane Katrina data, accuracy was so close to 1 even with a single format that including more formats yielded little additional benefit.

Figure 45. Accuracy of reusing topes on Hurricane Katrina (HK) and Google Base (GB) data.



Incidentally, the Hurricane Katrina 3B line in Figure 45 highlights the importance of asking users to double-check questionable values. As in Section 5.2.2, condition 3B accepts questionable values without double-checking them. As formats were added in this Hurricane Katrina experiment, topes started recognizing unusually-formatted, invalid strings that did not match the first couple of formats. The soft constraints in the topes identified these unusual strings as questionable, which causes condition 3A to reject them (correctly) and prompts condition 4 to simulate asking a user for double-checking (also correctly rejecting the strings). But condition 3B simply accepts the strings (since it accepts questionable strings). Normally, adding formats to validation code improves the quality of validation because it allows unusual-yet-valid strings. The lesson is simply adding formats to validation code can actually harm the accuracy of validation, if it doing so starts to allow unusual-and-invalid strings.

7.2.2 Scalability of example-based recommendation

To evaluate the responsiveness and accuracy of the tope recommendation algorithm as the number of topes increases, I ran simulations using Toped^{++}-based topes for all 32 most common kinds of data in the EUSES spreadsheet corpus (as identified by the procedure described by Section 5.2.2). Simulating what would happen if a user had a certain number of topes, I randomly chose a subset `T` of topes from these 32. For each tope, I 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, I retrieved the recommended topes. I then checked if the correct tope appeared in the result set. I computed the average recall and query time for different values of `|T|`, `|E|`, and number of results `N`. I then repeated the same process without using keywords.

Searching based on examples (even without keywords) provided a huge improvement over searching based on keywords alone, raising recall from 0.44 to as high as 0.65 even when the algorithm returned only a single search result (Figure 46). When the algorithm returned 5 search results, keyword-based search was still only 0.52, but keyword-based recall was as high as 0.83. The best recall resulted for `|E|=4` examples. 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 47), suggesting that in practice, it is best to limit the number of examples passed to the recommender. Including keywords slightly improved recall. Since recall rose substantially with N, it is desirable to return multiple options rather than simply assuming that the first is best.

Figure 46. Accuracy of the search-by-match tope recommendation algorithm showing that keywords and |E|=4 examples gave maximal recall. In this simulation, the number of topes |T| = 32.

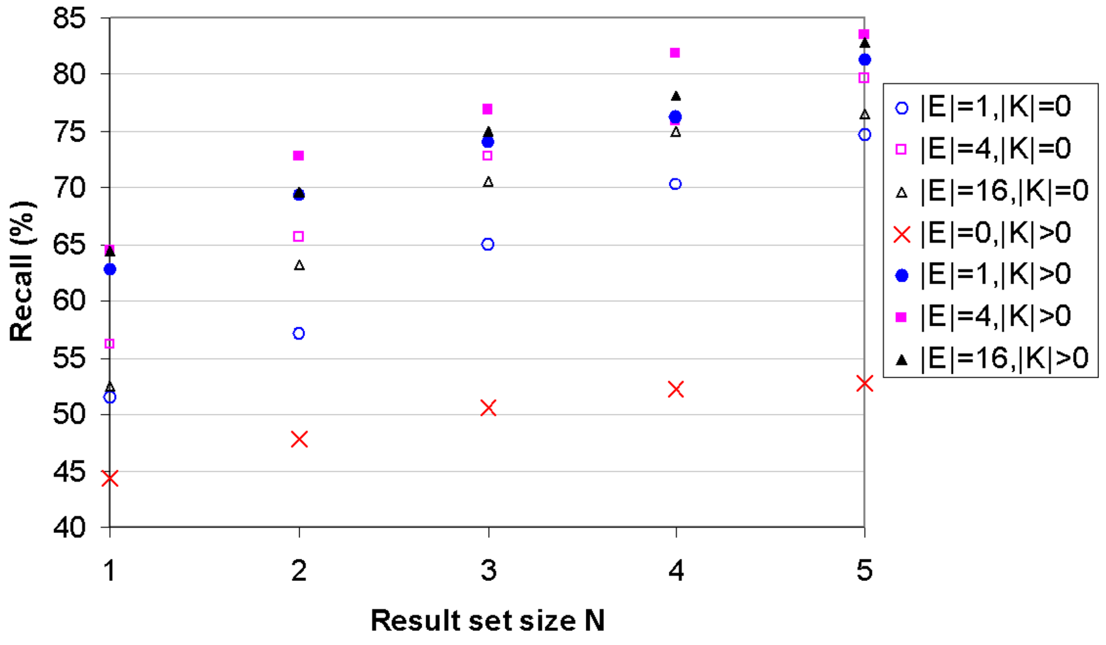
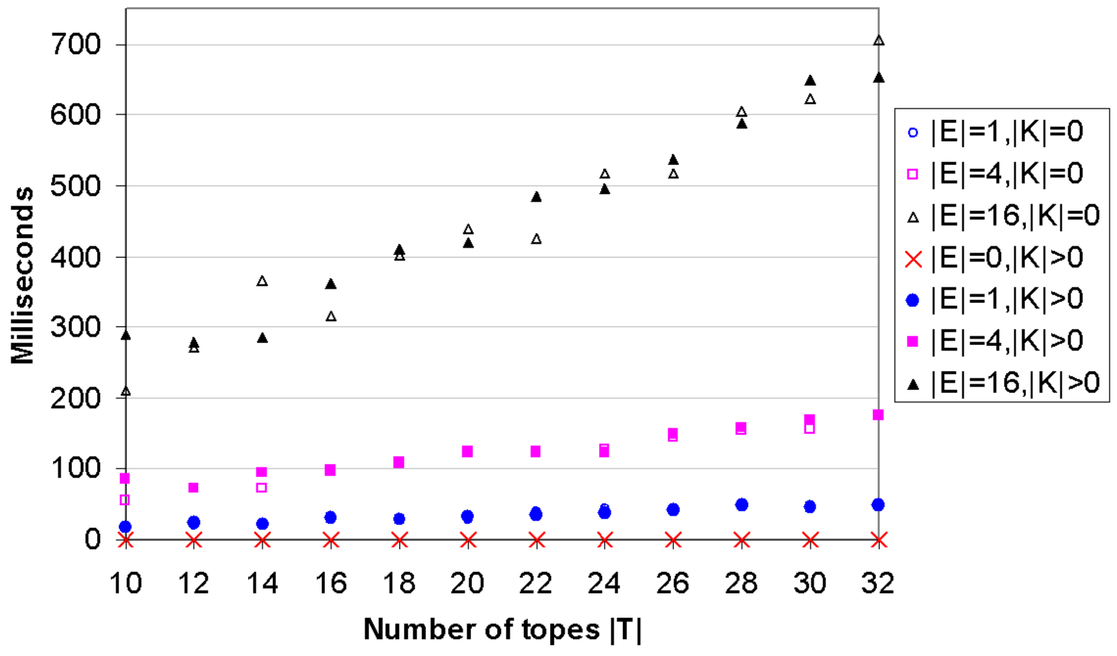


Figure 47. Speed of the search-by-match tope recommendation algorithm showing that recommendations were computed in under 1 s. In this simulation, the number of search results N=5.



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 noted earlier in this chapter, users are likely to have a few dozen topes on their computer (perhaps including tope implementations for the most common 32 data categories, as well as some organization-specific tope implementations). Searching a collection of 50 topes would require approximately 250 ms, while searching through 200 topes would require just under 1 sec. Instrumenting the code showed that the CharSig optimization was able to rule out approximately half of all topes during queries.

Scaling up the search-by-match algorithm through parallelization

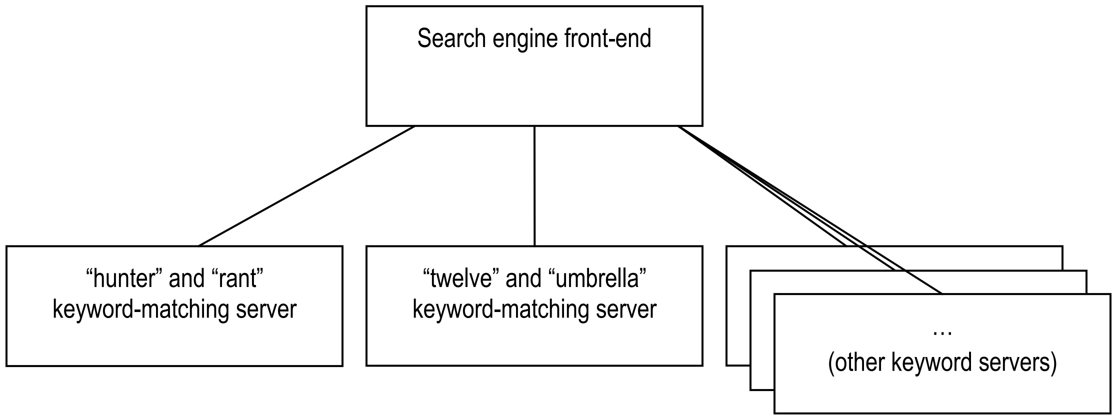
The analysis above showed that the search-by-match algorithm is fast enough to search through a couple hundred topes in under a second. This probably is fast enough for searching a small organization-sized repository. However, to search through all the topes that a large organization might produce, or to search through thousands of topes produced by members of an entire society, the algorithm would need to be parallelized. This section describes one straightforward approach (that has not been implemented) for parallelizing the algorithm *nearly completely*, showing the feasibility of scaling the search-by-match algorithm to support extremely large repositories.

The search-by-match algorithm has three parts, each of which lends itself to nearly total parallelization. These three parts are the index-based search (for keywords and whitelist matches), the tests of examples against topes (using CharSigs then `isa` functions), and periodic pruning (for sorting and filtering results).

The keyword and whitelist portions of the algorithm have the general structure of receiving a list of literal strings (keywords and example strings, respectively), then returning a list of tope implementations with the number of "hits" for each tope implementation. This is precisely the same algorithmic structure embodied by a standard web search engine, which accepts a set of user-specified keywords and returns a list of documents with the number of hits for each document. Companies like Google have been very successful at parallelizing this kind of algorithmic structure by partitioning the index over tens of thousands of computers [7].

For example, one computer could be responsible for returning documents that match the keywords "hunter" and "rant", while another could be responsible for returning documents that match the keywords "twelve" and "umbrella" (Figure 48). To improve load balancing, keywords are randomly assigned to pools (which is why two totally unrelated words like "hunter" and "rant" might be served by the same pool). When a computer acting as a front-end for the search engine receives a query such as "umbrella hunter", it calls the appropriate server for each keyword to ask which documents contain the specified keyword, and the appropriate server reports a list of documents which front-end combines into a result set. A similar parallelization structure is used to satisfy more sophisticated search filters (as in queries requiring that two keywords appear adjacent to one another in the source document).

Figure 48. Basic search engine parallelization structure


Search engine front-end
"hunter" and "rant" keyword-matching server
"twelve" and "umbrella" keyword-matching server
…
(other keyword servers)

As an implementation detail aimed at avoiding poor throughput, intolerable latency, and unreliability in the event of hardware malfunctions, each box in Figure 48 actually corresponds to a load-balanced pool of replicated computers rather than a single computer. Moreover, the parallelization tree might have more than two levels, in order to pre-filter results prior to combination by the front-end pool. High-speed network switches connect the computers, which typically reside in the same rack of a data center.

This parallelization strategy would work well for the keyword phase and the whitelist phase of the first part of the search-by-match algorithm. In the first phase, a tree of computers would find tope implementations (rather than documents, as in web search) that match certain keywords. This would yield a list of tope implementations sorted in order of how many keywords they match. Then a second tree could find tope implementations using an index mapping literal examples to tope implementations that contain those strings in their respective whitelists. When bubbling the results up this second tree, the results would first be sorted by the number of keyword hits that were reported by the first phase of the search, and whitelist hits would only be used as tie-breakers. This would yield a sorted list of search results based on the desired combination of keyword and whitelist matches.

A very similar parallelization strategy could be used for the second main part of the search-by-match algorithm, finding tope implementations based on how well they match the examples. This second part would be divided into two phases, each requiring a parallelization tree.

In the CharSig phase, each computer in a tree would be responsible for certain CharSigs, rather than certain keywords (Figure 49). For example, one computer might be responsible for returning tope implementations that are in S^5_{letter} or S^2_{digit}, while another computer might be responsible for returning tope implementations that are in S^7_{alnum} or S^1_{-}. When faced with an example like “ABC-DE20”, the front-end would intersect results from these different computers to find topes that match $S^5_{\text{letter}} \cap S^2_{\text{digit}} \cap S^7_{\text{alnum}} \cap S^1_{-}$, yielding a short list of tope implementations.

The next phase would refine this prospective list of tope implementations by testing each example `s` against each tope implementation to determine the maximal `isa(s)`. This could be parallelized by partitioning the set of known tope implementations in the repository among another pool of computers in a (fourth) tree of computers (Figure 50). For example, tope implementations could be assigned to servers based on their GUID `mod 10`. When bubbling the search results up this tree, the results would first be sorted by the number of keyword hits, with whitelist hits as a tie-breaker, and with maximal `isa` scores acting as a final tie-breaker.

Figure 49. CharSig parallelization structure

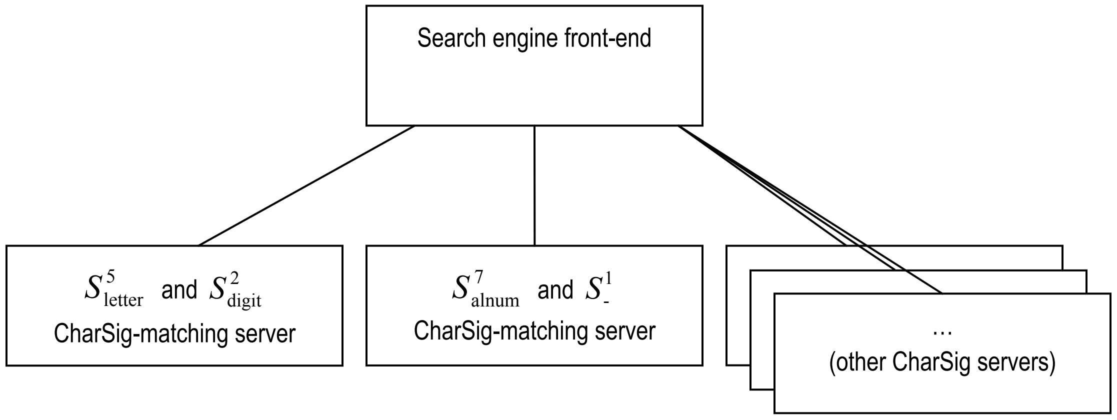
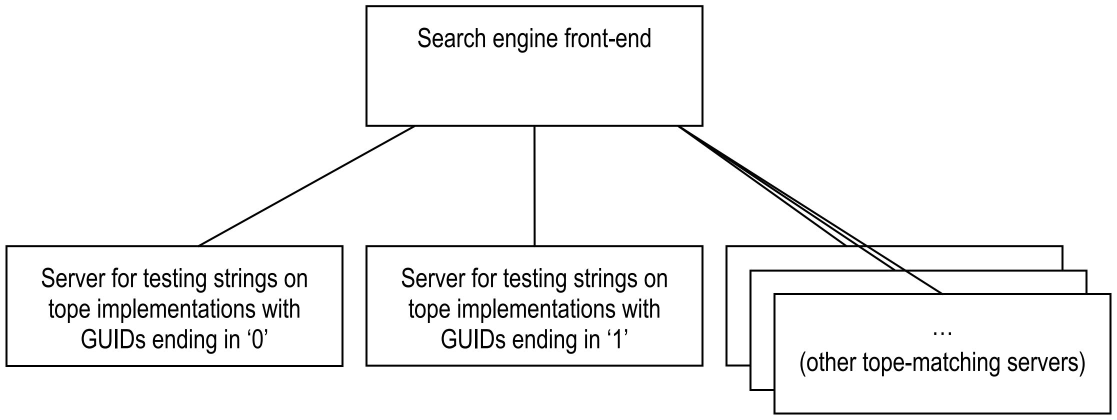


Figure 50. `isa`-matching parallelization structure


Search engine front-end
Server for testing strings on tope implementations with GUIDs ending in '0'
Server for testing strings on tope implementations with GUIDs ending in '1'
…
(other tope-matching servers)

The last part of the algorithm, pruning, actually occurs at several points along the algorithm. This part of the algorithm ensures that at any point in time, no tope implementations remain under consideration if they could not be part of the final set of `N` search results. The front-end server could perform this step between calls to each of the parallelization trees described above.

In short, the tope recommendation algorithm could be parallelized with a combination of four trees: one for keyword-matching, one for whitelist-matching, one for CharSig-matching, and one for `isa`-matching. The first two trees would look essentially identical to standard search engine parallelization trees. The CharSig-matching tree would map CharSigs to tree leaves (instead of mapping keywords to tree leaves), and the `isa`-matching tree would map tope implementations to tree leaves. In a relatively small deployment, a single front-end server could act as the tree root for each of these trees, or as in the case of larger search engine systems, a multi-level tree could be used to perform pre-filtering prior to combining search results. Having put a parallelization structure in place, each node in each tree could then be implemented with a load-balanced replication pool in order to provide adequate levels of scalability.

Chapter 8. Filtering data descriptions based on reusability traits

Different tope implementations will not be equally reusable, even when they are intended to describe the same data category. For example, ten data descriptions might match every single example provided to the search-by-match algorithm, yet some might be much easier to reuse than others. One reason for varying reusability is that the people who create the tope implementations might have different skill levels, so some implementations might have some errors. Another reason for varying levels of reusability is that some people might design certain tope implementations for very specific purposes (perhaps only matching a few of a data category's formats), rather than for general reuse. Conversely, some tope implementations on a repository might only contain very generic constraints, while a user might need a tope implementation that contains specific specialized constraints (such as a tope implementation for Carnegie Mellon University phone numbers, which must obey more constraints than just any phone number).

Consequently, it would be helpful if the repository supplemented the search-by-match algorithm with additional features for identifying tope implementations that are likely to be reusable. In particular, the topes repository provides features for filtering the search results to identify those that are probably reusable. (Even better would be to allow the user to *sort* the search results based on a combination of reusability criteria and search-by-match criteria, but this is reserved for future work.)

But tope implementations are hardly the first kind of reusable end-user code. Other examples of sometimes-reused end-user code include web macros [20], Scratch animations [94], AgentSheets (simulations written in a visual programming language) [117], Matlab files [38], spreadsheets [121], VisualBasic [123], and HTML [98].

Thus, the need for reusability-oriented search features is not unique to the topes repository. Many repositories for end-user code successfully meet similar requirements. One example is the coscripter.com web macro repository, which can sort web macros in the search results based on how often they have previously been downloaded, as an indication of the reusability of each macro. Another example is the AgentSheets repository for sharing simulation code, which allows users to filter search results based on the code's hardware requirements (which are specified by the code author in an annotation). A third example is the mathworks.com repository for sharing Matlab code, which allows users to filter search results to only include code that has an "M-file" structure allowing the code to be reused by running it from the command-line (or from another Matlab script). This search filter essentially helps a user to find code that he can try out even before reading the code: he can simply run the code by double-clicking on its M-file's icon in Windows, then can inspect the results to see if they meet his needs. This allows him to understand and evaluate the code's function without first having to read the actual text of the code.

Each of the search features described above is driven by one trait of the code stored in the corresponding repository. In the case of CoScripter macros, the trait is indirect and automatically collected by the repository: a record of how often each macro has been downloaded. The AgentSheets filter is driven by information about hardware requirements that is manually provided by the publisher in the form of an annotation. The Mathworks M-file filter is based on a trait that is directly computed from the code itself, reflecting the structure of the source file.

None of these traits are guarantees of reusability, as each provides evidence about only one dimension of reusability. The number of web macro downloads is an indication of how broad a macro's appeal seems to be, such that users consider it worth downloading to try out (with no further indication about other dimensions of reusability, such as whether the users were actually satisfied after downloading a macro). The AgentSheets hardware annotation indicates whether a piece of simulation code can be used on a broad range of different computers, which is something of an indication about the compatibility or flexibility of the code to different execution contexts, but it does not address the dimension of mass appeal hinted at by the number of downloads. The Mathworks M-file trait reflects how easy it is for users to understand the code by trying it out, but this trait in turn tells nothing about mass appeal or compatibility / flexibility. Thus, while each trait provides a limited amount of evidence of reusability, it is not proof of reusability.

Despite the limited evidentiary content of these traits, and despite the fact that each repository provides only a few search features based on such traits, each repository's search engine has sufficed for helping thousands of users to share and reuse code. In general, these search features do not take advantage of more sophisticated traits that researchers have identified for finding reusable code created by *professional* programmers. For example, Basili et al have found that certain metrics computed on object-oriented code give a very good estimate of how likely it is that the code contains defects [8], which perhaps might give an insight about reusability if defects interfere with code reuse. Researchers have even begun to incorporate simple object-oriented metrics into features that search through repositories for highly reusable code created by professional programmers [5].

To sum up, *a few simplistic* reusability-oriented search features have generally sufficed in repositories for *end users*, yet researchers are actively investing a great deal of effort into developing *more sophisticated* search features for repositories aimed at *professional* programmers. This state of affairs raises several questions.

First, if we were similarly to develop more sophisticated search features for repositories aimed at end users (such as the topes repository), what reusability traits should those features rely on? Put another way, what are the traits of reusable end-user code? Section 8.1 addresses this question by examining a wide range of empirical studies conducted by a variety of research groups, thereby identifying several kinds of traits that generally characterize highly reusable code created by end-user programmers. Many of these traits can be computed directly from the code, therefore requiring no additional investment of effort from any end users beyond uploading code to the repository.

Second, how much incremental benefit is gained by supporting increasingly complex reusability-oriented search features? As mentioned before, most repositories for end-user programmers have proven quite successful despite being fairly minimalist, offering only a few reusability-oriented search filters. Would search features based on reusability traits actually help an end-user repository to accurately identify reusable artifacts? Section 8.2 addresses this question in the context of one particular kind of end-user program, web macros, where a large user community already exists. Using information from the CoScripter repository, this analysis identifies traits that statistically correspond to higher levels of reuse. Further analysis shows that combining these traits through a machine learning model can accurately identify reusable code, though with generally diminishing returns after using more than 6 to 11 traits.

This result suggests that although a handful of simple reusability-based search filters generally suffice for many repositories aimed at end users, it may be desirable to include a few more filters. In particular, the presence of additional search filters could help users to identify reusable tope implementations. Section 8.3 presents the tope repository's "Advanced Search" set of twenty reusability-oriented search filters, from which users could select a desired subset of traits that should be demonstrated by tope implementations in search results.

8.1 Traits of reusable end-user code in general

Since tope implementations are a variety of end-user code, new tope repository search features are motivated by considering what traits are typically manifested by reusable end-user code in general. A variety of empirical studies provide a starting point for understanding what traits characterize reusable code created by end-user programmers.

These studies include the following:

- Retrospective analyses of the CoScripter web macro repository [20]
- Interviews of software kiosk designers [21]
- A report about running a Matlab code repository [38]
- Observations of college students in a classroom setting [50]
- An ethnography of spreadsheet programmers [78]
- Observations of children using programmable toys [86]
- Interviews [98] and a survey of web developers [124]
- Interviews of consultants and scientists [112]
- Interviews of K-12 teachers [123]

These empirical studies are supplemented by one simulation of end-user programmer behavior [15], as well as empirical work related to professional programmers [13][24][74][99]. In the sub-sections below, citations of work related to professional programmers are underlined in order to make it clear when a statement is only supported by research on professionals.

Most of the traits uncovered by the empirical studies are reflected in the code itself (see Table 14). For example, the empirical studies suggest that for a given end-user programming environment, understandability seems to strongly affect reusability. (In particular, code with comments tends to be more reusable than code without comments because code lacking comments is hard to understand.) These traits could be called "direct" traits, since they are inherent properties of the code or other annotations directly provided in the code by the author.

A few reusability traits mentioned by empirical studies are indirect—not reflected in the code. In particular, these traits are reflected in information about the author and the code's prior uses. For example, the Matlab repository allows users to rate scripts on the repository, and its search engine allows users to sort search results based on average rating. The repository's designers report that, empirically speaking, they find that this feature helps users to successfully identify code that they can reuse [38]. In this case, the trait of reusable code is a high rating by end users. The user interface is designed to reflect the fact that that scripts with this trait are more reusable than scripts that lack the trait (having low ratings instead).

The next sub-sections discuss the direct and indirect kinds of reusability traits in more detail. A recurring theme throughout these sub-sections is that these traits do not guarantee reusability. For example, even if a web macro has comments and has been downloaded many times, this does not guarantee that it will be reusable to a particular programmer. Even when taken together, these traits are *evidence* of reusability, not proof.

Table 14. Traits of reusable code, as uncovered by empirical studies of end-user and professional programmers.

Traits reflected in…	End users											Professionals			
	[15]	[20]	[21]	[38]	[50]	[78]	[86]	[98]	[112]	[123]	[124]	[13]	[24]	[74]	[99]
Code															
Mass appeal					x						x	x			x
Flexibility		x			x		x	x		x	x	x			
Functional size	x											x	x	x	
Understandability			x		x	x		x	x	x		x			x
Code's authorship						x									
Code's prior uses				x											

8.1.1 Traits directly reflected in end-user code

It is widely believed that professional programmers' code is more reusable if it meets four criteria [13]: the code should be functionally *relevant* a wide variety of programmers' problems, it should be *flexible* enough to meet those varying requirements without requiring a great deal of modification of existing code, it should be *functionally large* enough to justify reuse without involving substantial extension by programmers, and it should be *understandable* to the people who reuse it.

Meeting these four requirements also apparently contributes to the reusability of end-user programmers' code, since the eleven end-user programming studies produced many findings of the form, "Code was hard to reuse unless it had X," where X was a trait suggesting that the code met one of the four requirements. For example, unless code contained comments, teachers had difficulty understanding it and therefore had trouble reusing it [123]. In this example, the trait is the presence of comments in the code and the requirement is understandability.

Traits suggesting that the code has mass appeal / functional relevance to a wide audience

When programmers search a repository, they want code that meets some functional need in the context of the programmer's work [24][50][99]. Typically, only a small amount of code is functionally relevant to many contexts, so a simple functional categorization of code can be evidence of its reusability. For example, 78% of mashup programmers in one survey created mapping mashups [124]. All other kinds of mashups were created by far fewer people. Thus, just knowing that a mashup component was related to mapping (rather than photos, news, trivia, or the study's other categories) suggested mass appeal. In addition, since repositories and programming environments usually provide a keyword-based search interface, code that matches popular keywords tends to be returned more often in search results and apparently reused more often than code that does not match these popular keywords [50]. Certain keywords, therefore, can suggest functional relevance to a broad audience.

Traits suggesting that the code has flexibility to support reuse

Reusable code must not only perform a relevant function, but it must do it in a flexible way so that it can be applied in new usage contexts. Flexibility can be evidenced by use of variables rather than hardcoded values. In a study of children, parameter-tweaking served as an easy way to "change the appearance, behaviour, or effect of an element [component]", often in preparation for composition of components into new programs [86]. Web macro scripts were more likely to be reused if they contained variables [20].

Flexibility can be limited when code has non-local effects that could affect the behavior of other code. Such effects reduce reusability because the programmer must carefully coordinate different pieces of code to work together [50][123]. For example, web page scripts were less reusable if they happened to "mess up the whole page" [98], rather than simply affecting one widget on the page. In general, non-local effects are evidenced by the presence of operations in the code that write to non-local data structures (such as a web page's entire document object model in the case of reusing a web page script).

Finally, flexibility can be limited when the code has dependencies on other code or data sources. If that other code or data become unavailable, then the dependent code becomes unusable [124]. Dependencies are evidenced by external references reflected directly in the source code. For example, users were generally unable to reuse web macros that contained operations which read data from intranet sites (i.e.: sites that cannot be accessed unless the user was located on a certain local network) [20].

Traits suggesting that the code has adequate functional size to justify reuse

When asked about whether and why they reuse code, professional programmers made "explicit in their verbalisation [sic] the trade-off between design and reuse cost" [24], preferring to reuse code only if the effort of doing so was much lower than the effort of implementing similar functionality from scratch. In general, larger components give a larger "payoff" than smaller components, with the caveat that larger components can be more specialized and therefore have less mass appeal [13]. Empirically, components that are reused tend to be larger than components that are not reused [74].

Simulations suggest that end-user programmers probably evaluate costs in a similar manner when deciding whether or not to reuse existing code [15], though there do not yet seem to be any interviews or other empirical studies showing that end-user programmers evaluate these costs consciously.

Traits suggesting that the code is understandable

Understanding code is an essential part of evaluating it, planning any modifications, and combining it with other code [50]. Moreover, understanding existing code can be valuable even if the programmer chooses not to directly incorporate it into a new project, since people often learn from existing code and use it as an example when writing code from scratch [98][99]. This highlights the value of existing code not only for verbatim blackbox or near-verbatim whitebox reuse, but also for indirect conceptual reuse.

Many studies of end-user programmers have noted that understandability is greatly facilitated by the presence of comments, documentation, and other secondary notation. Scientists often struggled to reuse code unless it was carefully documented [112], teachers' "comprehension was also slow and tedious because of the lack of documentation" [123], and office workers often had to ask for help in order to reuse spreadsheets that lacked adequate labeling and comments [78]. End-user programmers typically skipped putting comments into code unless they intended for it to be reused [21]. In short, the presence of comments can be strong evidence of understandability and, indirectly, of reusability.

8.1.2 Indirect reusability traits reflected in end-user code's authorship and prior uses

A few studies suggest that the identity of code's author can provide evidence about the code's reusability. For example, in some organizations, certain end-user programmers have been tasked with cultivating a repository of reusable spreadsheets; in these organizations, just knowing who wrote a spreadsheet provides some information about its reusability [78]. In another study, scientists reported that they only tended to reuse code if it was written by certain other scientists [112].

Once someone has tried to reuse code, recording that person's experiences can capture information about the code's reusability. Repositories of end-user code typically record this information as reviews, recommendations, and ratings [38][64]. In the Matlab repository, capturing and displaying these forms of reusability evidence has helped users to find high-quality reusable code [38].

8.2 Evaluation: Effectiveness of using traits to identify reusable end-user code

I performed an in-depth analysis of logs from the CoScripter repository in order to empirically evaluate how well the traits of end-user code actually could identify reusable code. The evaluation proceeded in several stages.

First, an extract of logs for a 6-month time period was retrieved from the repository server, as well as the source code of the first version of each web macro that was created or executed during this time period. Second, for each macro, 35 traits were computed. Each trait was selected because it seemed like a plausible indicator of reusability based on the analysis described by Section 8.1. Third, four measures of reuse were computed for each web macro. Fourth, for each combination of trait and reuse measure, a statistical test was computed to determine whether the trait actually statistically corresponded to that measure of reuse; this provided a general picture of how strongly each trait was related to reuse. Finally, for each measure of reuse, a simple machine learning model was trained on the traits to predict whether each macro would be reused.

These experiments show that the selected traits do indeed contain enough information to fairly accurately predict whether a web macro will be reused. In fact, using as few as 6 to 11 traits provides enough information to predict reuse with 70-80% recall (at 40% false positive rate).

8.2.1 Suitability of the CoScripter repository as a setting for the evaluation

There are three reasons why the CoScripter web macro repository is an appropriate setting for this evaluation of the effectiveness of using traits to identify reusable end-user code.

First, with macros provided by nearly 10,000 users over the past two years, this repository offers an insight into the dynamics of reuse once a user community grows beyond the tiny number of people commonly involved in research studies evaluating new end-user programming tools. As researchers and colleagues, the developers of CoScripter at IBM have been generous in providing historical data, which is typically difficult or impossible to obtain about commercial programming tools once they leave the research stage. As a result, the CoScripter repository offers a uniquely realistic insight into what happens in large communities of end-user programmers.

Second, CoScripter macros and tope data descriptions are alike in several ways that make studying reuse of web macros a sensible surrogate for studying reuse of data descriptions (in the absence of a large user base for topes). For each of the four primary requirements driving reusability (summarized by Section 8.1.1), data descriptions and web macros have key similarities:

- *Understandability:* Both web macros and data descriptions can be annotated with comments, which was by far the most oft-cited understandability trait mentioned in the empirical studies.
- *Functional size:* CoScripter and data descriptions have comparable visual complexity, typically requiring a few lines of information on the screen (with lines corresponding to constraints in data descriptions and executable instructions in web macros). These lines of information offer a measure of functional size for each kind of code.
- *Flexibility:* For both web macros and data descriptions, the primary approach for supporting flexibility is to allow "sloppiness". Specifically, CoScripter makes a best-effort attempt to interpret macro code in the context of a web page, while the TDE supports soft constraints that might not be perfectly satisfied at runtime.
- *Mass appeal:* The primary search screens of CoScripter and the TDE allow users to search the respective repository based on keywords, which might be indicative of code's mass appeal.

The third and most important reason why the CoScripter repository is an appropriate setting for this evaluation is that, like the reuse of tope implementations, the reuse of web macros is fundamentally predicated on the fact that end-user programmers often have certain kinds of data in common. Consider: the reason why a CoScripter script is reusable at all between two people is because those two people have a reason to access the same web sites and perform the same operations on those web sites. Those operations manipulate web form widgets, so two people who use the same web macro not only use the same web site, but they work with exactly the same web form widgets (often text fields and drop down lists). If two people need to manipulate the same web form widget, then they both need to work with the same kind of data—whatever the data in that widget happens to be. Thus, CoScripter reuse is fundamentally predicated on the assumption that end users have certain kinds of data in common. That commonality is the fundamental bedrock that ultimately motivates inter-person reuse of tope implementations, as explained in Chapter 7.

Now, CoScripter reuse is also predicated on a variety of additional assumptions, and the TDE "knocks down" many of these assumptions. In particular, CoScripter macro reuse requires users to not only have a data category in common, but they also must be working in the same application tool (Firefox) with data from the same source (a certain web site), and it must be acceptable for the macro source code to be publicly visible (as coscripter.com is open to all users). In contrast, tope implementations can be reused even when users are not working in the same application tool (as tope implementations are application-agnostic) and when they are not working with data from the same source (as a tope can be reused to validate strings in multiple web applications, spreadsheets, and so forth). Moreover, data descriptions can be kept private to organizations (as users can register organization-specific repositories).

These additional assumptions probably reduce the potential for reuse in the CoScripter context, making it harder for users to find reusable web macros. Indeed, one study of the CoScripter repository found that only 9% of all web macros were ever executed by more than three users [20]. Finding a reusable web macro can be like looking for a needle in a haystack.

Yet as described below, reusability-oriented web macro traits still contain enough information to find those needles automatically. This result suggests that analogous traits will suffice for identifying reusable code in situations where reuse is more commonly feasible because reuse is predicated on fewer assumptions. In particular, if reusability-based search would work for the CoScripter repository, then it also seem likely to work for the topes repository.

8.2.2 Web macro traits: raw materials for a model

For this analysis, I selected web macro traits based on the general requirements for reuse as outlined by Section 8.1.1. Specifically, users need to be able to find functionally relevant macros (with more widely reused macros probably being those with more mass appeal), macros need to be flexible enough to be reused with minimal modification, macros need to contain enough functionality so that users do not need to modify them much to add missing functionality, and users need to be able to understand macros well enough to choose appropriate macros and make any necessary modifications. These identified traits were supplemented with traits based on information about authorship. No traits were considered that required information about prior uses of macros (since relying on such traits would have limited the applicability of the evaluation's results to situations in which macros had previously been used). In total, these considerations led to the identification of 35 traits in the following 5 categories (summarized in Table 15):

- *Mass appeal*: Macros might be more likely to be found if they are relevant to the interests of the CoScripter community, as reflected by website URLs and other tokens in macros. Promotion of a macro as a tutorial (on the homepage) might suggest mass appeal, since CoScripter tutorials are designed to be of general interest to many users.
- *Flexibility*: Parameterization and use of mixed-initiative instructions might increase the flexibility of macros, reducing the need for modification. A prior study found that reuse seemed lower for macros with preconditions such as requiring the browser to be at a certain URL prior to execution, requiring the user to be logged into a site prior to execution, or requiring that many sites are online during execution [20]. Extra effort might be needed to understand or modify macros with preconditions.
- *Functional size reflected in length*: Longer code requires more effort for understanding, and many studies indicate that longer code tends to have more defects [32] that might require modification prior to reuse. Counterbalancing these factors, longer macros contain more functionality, which often increases the value of reuse.

Table 15: Each row shows one macro trait tested for statistical correspondence to four measures of reuse. Traits are sorted in the order of introduction in Section 8.2.2. A + (−) hypothesis Hyp indicates that I expected higher (lower) levels of reuse to correspond to the characteristic. **Non-empty empirical results** indicate statistically significant differences in reuse, with + (−) indicating that higher (lower) reuse corresponds to higher levels of the characteristic. One + or − indicates one-tail significance at p<0.05, ++ or −− indicate p<0.01, and +++ or −−− indicate p<0.00036 (at the Bonferroni correction of p<0.05). Shaded cells were particularly useful in the machine learning model (Section 8.2.5).

Trait				**Empirical Results**			
Category	**Name**	**Meaning**	**Hyp**	**Self Exec**	**Other Exec**	**Other Edit**	**Other Copy**
Mass appeal	*keycnt*	real: normalized measure of how many scripts contain the same tokens as this script	+	+			
	keydom	real: normalized measure of how many other scripts contain the same URL domains as this script	+		+++	++	+
	nmchost	int: # URLs in script that use numeric IP addresses	−				
	niinet	int: # hosts referenced by script that seem to be on intranets	−		−		
	tutorial	bool: true if script was created for tutorial list	+		+	+++	+++
Flexibility (including preconditions)	*npar*	int: # parameters (configuration variables) read by script	+	++	+	+++	+++
	nlit	int: # literal strings hardcoded into script	+	+++			
	nyloc	int: # mixed-initiative "you manually do this" instructions	+		+	+	++
	assmurl	bool: true if first line of script is not a "go to URL" instruction	−		−−−		
	ctnclstt	bool: true if script contains "log in", "logged in", "login", or "cookie"	−				
	nhosts	int: # distinct hostnames in script's URLs	−		+++		
Func. Size	*sloc*	int: total # non-comment lines in script	−	++			
	tloc	int: total # lines (*sloc* + *cloc*)	−	+			
	ndloc	int: total # distinct non-comment lines in script	−	+++			++
Understandability: language and annotations	*enus*	int: # US URLs in script	+	+	+		
	nonenus	int: # non-English words in literals + # of URLs outside USA	−	−		−	−−
	unklang	bool: true if *nonenus* and *enus* are each 0	−		−−−		
	nonroman	pct: % of non-whitespace chars in title or content that are not roman	−		−	−	
	cloc	int: # comment lines	+	+	++	+++	+++
	titlts	bool: true if script title contains the word "test"	−	−−			
	titlcp	bool: true if script title contains the phrase "Copy of"	−	−−	−		
	titled	bool: true if script has a title	+	+++	++		
	titlpn	bool: true if script title contains punctuation other than periods	−	−			
Authorship	*aid*	int: id of the user who authored the script (lower for early adopters)	−	+++	−−−	−−	−−−
	sid	int: id of the script (tends to be lower for early adopters)	−		−−−	−−−	−−−
	ibm	bool: true if script's author was at an IBM IP address	+	++		+++	+++
	nforum	int: # posts by the script author on the CoScripter forum	+	++		++	
	aulonaname	bool: true if script author's name starts with punctuation or 'A'	+		−		
	npcreated	int: # scripts by same author created prior to this script	+	+++	−−−		
	npselfexec	int: # scripts by same author executed by author prior to this script's creation	+	+++	−−−	−−	
	npoexec	int: # scripts by same author executed by other users prior to this script's creation	+	−−−	+++	−	−
	npoedit	int: # scripts by same author edited by other users prior to this script's creation	+	−−−	+++	−	−
	npocopy	int: # scripts by same author copied by other users prior to this script's creation	+	−−−	+++	−	−
	ctnordin	bool: true if script uses ordinals (eg: "third") to reference fields	+	+++			
	ctlclick	bool: true if script uses "control-click" or "control-select" keywords	+		+		+++

- *Understandability*: Macros might be more understandable if their data and target web sites are written in the community's primary language (English). Code comments and proper macro titles might increase understandability.
- *Authorship*: Early adopters, IBM employees, active forum participants, and users who have already created many widely-reused macros might be more likely to produce macros that work properly and can be reused with minimal modification. If macros contain advanced keywords (control-click instructions, and ordinal widget references), this might suggest that their authors were experts who could produce macros that work properly without modification, though advanced syntax might inhibit understandability.[10]

8.2.3 Computing measures of reuse

The 6 months of web server logs extracted from the CoScripter web server recorded operations on 937 macros. Four measures of reuse were computed for each macro:

- *Self-exec*: Did the macro author ever execute the macro between 1 day and 3 months after creating the macro? (Executions within 1 day were omitted, since such executions could relate to the initial creation and testing of the macro, rather than reuse per se.) {17% of all macros met this criterion}
- *Other-exec*: Did any other user execute the macro within 3 months of its creation? {49%}
- *Other-edit*: Did any other user edit the macro within 3 months of its creation? {5%}
- *Other-copy*: Did any other user copy the macro to create a new macro within 3 months of the original macro's creation? {4%}

[10] It is debatable whether these last two traits really belong in the authorship category, as they are computed from the code to provide hints about authorship. An alternative would be to categorize them under Understandability, with a hypothesis that the traits will correspond to lower understandability. The absence of a perfect categorization highlights the fact that some traits might simultaneously have positive and negative effects on reusability.

Binary measures of reuse were more suitable than absolute numbers of reuse events for two reasons. First, unless a repository user chooses to sort macros by author, the user interface sorts macros by the number of times that each has been run. This appears to result in a “pile-on” effect (properly called an “information cascade” [14]): oft-run macros tend to be reused very much more in succeeding weeks. This interferes with using absolute reuse counts as a measure of reuse for testing macro traits. Second, macros can recursively call themselves (albeit without parameters), and some users also apparently set up non-CoScripter programs to run CoScripter macros periodically (e.g.: once per day), clouding the meaning of absolute counts.

Though no macros in the time interval were recursive, some ran periodically. When computing reuse measures, a macro was considered to have been reused if a periodic program ran it. However, a careful examination of the logs identified several automated spiders that walked the site and executed (or even copied) many macros in a short time, so those events were filtered. Most such spiders ran from IBM IP addresses, apparently owing to automated analyses by colleagues at IBM.

The 3 month post-creation observation interval was a compromise. Using a short interval runs the risk of incorrectly ruling a macro as un-reused, if it was only reused after the interval ended. Selecting a long interval reduces the number of macros whose observation interval fell within the 6 months of available logs. Selecting half of the log period as the observation interval resulted in few cases (20) of erroneously ruling a macro as un-reused yet yielded over 900 macros for study.

8.2.4 Testing macro traits for correspondence to reuse

For each trait, macros were divided into two groups based on whether each macro had an above-average or below-average level of the trait (for Boolean-valued traits, treating “true” as 1 and “false” as 0). For each group’s macros, I computed the four reuse measures. Finally, for each combination of trait and reuse measure, I performed a one-tailed z-test of proportions. In cases where the correspondence between macro trait and reuse measure was actually opposite the expected result, Table 15 reports whether a one-tailed z-test would have been significant in the opposite direction. The table shows statistical significance at several levels, including a level based on a Bonferroni correction that compensates for the large number of tests (140). Some macro traits are not statistically independent (e.g.: *sloc* rises with *tloc*), nor are the measures of reuse (e.g.: an *other-exec* event almost always preceded each *other-copy* and *other-edit* event). Thus, the correction is likely to be over-conservative and establishes a lower-bound on the statistical significance of results.

In terms of robustness, I noted that many candidate macro traits are count integers that could be normalized by the overall macro length. Consequently, I tested these traits twice, once with the traits shown in Table 15, and again with length-normalized traits as appropriate. In virtually every case, results were identical.

Results of statistical tests

As expected, many macro traits corresponded to reuse. Interestingly, different reuse measures corresponded to different traits, suggesting that different kinds of reuse occur for different reasons. Thus, rather than combining macro traits into one model that attempts to accurately predict all possible forms of reuse, we need a generalized model that can be instantiated for each measure of reuse.

The *Authorship* expertise traits did not correspond to reuse exactly as expected. Macros created by apparent experts were more likely to be run by other users, but less likely to be edited—perhaps such macros worked properly and rarely needed modification.

The *Functional Size / Length* traits corresponded to higher likelihood of reuse by the macro author, though this was not a complete surprise. Empirically, the increased functionality in longer macros seemed to outweigh any risk of increased defects: just as popularity is an imperfect proxy for reusability, defect-proneness need not correspond directly to reusability.

These statistical tests show it is possible to find traits that correspond to reuse and that are automatically computable when a macro is created. Thus, while this set is not a definitive list of all traits that might correspond to reuse, it can serve as a basis for developing a predictive model of macro reuse.

8.2.5 Prediction of reuse

The machine learning model predicts reuse based on how well a macro's traits satisfy a set of simple arithmetic constraints that I call "predictors". A machine learning algorithm selects constraints that predict the reuse measure used during training. For example, one predictor might be a constraint on the number of comments such as *cloc* ≥ 3, another might be that the number of referenced intranet sites *niinet* ≤ 1, and a third might be *titled* ≥ 1. Ideally, all such predictors will be true for every reused macro and false for every un-reused macro.

After the first algorithm selects a set of predictors, a second algorithm uses this set to predict if some other macro will be reused. It counts the number of predictors matched by the macro and predicts that the macro will be reused if the macro matches at least a certain number of predictors. Continuing the example above, requiring at least 1 predictor match would predict that a web macro will be reused only if *cloc* ≥ 3 or *niinet* ≤ 1 or *titled* ≥ 1.

Training and using the predictive model

The algorithm trains a predictive model in three stages (Figure 51). Its inputs are a training set of macros R', real-valued traits C defined for each macro, a measure of reuse m that tells whether each macro is reused, and a tunable parameter α described below. Its output is a set of predictors Q.

First, the algorithm determines if each macro trait c_i corresponds to higher or lower reuse. To do this, the algorithm places macros into two groups (R_m and $\bar{R}_m$), depending on if each macro was reused according to measure m. It compares the proportion of reused macros that have an above-average value of c_i to the proportion of un-reused macros that have an above-average value of c_i. If higher levels of c_i correspond to lower reuse, then the algorithm adjusts the trait by multiplying it by -1, so higher levels of adjusted traits a_i correspond to higher levels of reuse.

Second, for each adjusted trait a_i, the algorithm finds the threshold value τ_i that most effectively distinguishes between reused and un-reused macros. The algorithm selects the threshold that maximizes the difference between the proportion of reused macros that have an above-threshold level of adjusted trait versus the proportion of un-reused macros that have an above-threshold level of adjusted trait.

Finally, the algorithm creates a predictor for each adjusted trait that is relatively effective at distinguishing between reused and un-reused macros. As noted above, each predictor should ideally match every reused macro but not match any un-reused macros. Of course, achieving this ideal is not feasible in practice. Instead, a predictor is created if the proportion of reused macros that have an above-τ_i amount of the adjusted trait is at least a certain amount α higher than the proportion of un-reused macros that have an above-τ_i amount of the adjusted trait.

The minimal difference α between proportions in the reused and un-reused groups is a tunable parameter. Lowering α allows more adjusted traits to qualify as predictors, which increases the amount of information captured by the model but might let poor-quality predictors enter the model. This is a typical over-training risk in machine learning [72], so it is necessary to evaluate the empirical impact of changing α.

After training the model, predicting whether a new web macro will be reused requires testing each predictor on the macro (Figure 52). If the macro matches at least a certain number of predictors β, then the model predicts that the macro will be reused. This minimal number of predictors β is tunable. Lowering β increases the number of macros predicted to be reused, decreasing the chance that useful macros slip by. However, lowering β risks erroneously predicting that un-reused macros will be reused. As with α, this leads to a trade-off typical of machine learning, but in the case of β, the trade-off is directly between false negatives and false positives, rather than between wasting information and over-training. As with α, it is necessary to evaluate the empirical impact of changing β.

Figure 51. Algorithm TrainModel for selecting predictors based on training data

```
TrainModel
  Inputs: Training macros $R' \subseteq$ Repository $R$,
          where R is a set of macros
        Macro traits $C = \{c_i : R \rightarrow [0, \infty)\}$
        Measure of reuse $m : R \rightarrow \{0,1\}$
        Minimal proportion difference $\alpha \in (0,1)$
  Outputs: Predictors $Q = \{ q_i : R \rightarrow \{0,1\} \}$

Let the reused macro set $R_m = \{s \in R' : m(s) = 1\}$
Let the un-reused macro set $\overline{R}_m = \{s \in R' : m(s) = 0\}$
Let $p(S, c, \tau) = \left|\{s \in S : c(s) \geq \tau\}\right| / \left|S\right|$

Initialize Q to an empty set of predictors
For each $c_i \in C$ ,
  Let $\mu_i = \sum_{s \in R'} c_i(s) / \left|R'\right|$
  Let adjusted trait
    $a_i(s) = c_i(s)$ if $p(R_m, c_i, \mu_i) \geq p(\overline{R}_m, c_i, \mu_i)$
    or $a_i(s) = - c_i(s)$ otherwise
  Compute threshold (through exhaustive search)
    $\tau_i = \text{argmax } p(R_m, a_i, \tau) - p(\overline{R}_m, a_i, \tau)$
  If $p(R_m, a_i, \tau_i) - p(\overline{R}_m, a_i, \tau_i) \geq \alpha$
    then add the following predictor to Q…
    $q_i(s) = 1$ if $a_i(s) \geq \tau_i$ and 0 otherwise
Return Q
```

Figure 52. Algorithm EvalMacro for predicting whether a macro will be reused

```
EvalMacro
  Inputs: One macro $s \in$ Repository R
        Minimal predictor matches $\beta \in (0, \left|Q\right|]$
        Predictors $Q = \{ q_i : R \rightarrow \{0,1\}\}$
  Outputs: Prediction of reuse $\in \{0,1\}$

Let nmatches = 0
For each $q_i \in Q$
  If $q_i(s) = 1$ then nmatches = nmatches + 1

If nmatches ≥ β then return 1 else return 0
```

Design decisions behind this model

Designing this model involved a number of choices that deserve further evaluation.

First, when selecting predictors, TrainModel chooses a threshold based on differences in proportions rather than commonly used entropy-based measures [72]. This decision seemed sensible because difference-in-proportions corresponds more directly than difference-in-entropy to the principle that ideal predictors match all reused macros but no un-reused macros. Evaluating this decision's impact (below) showed that the selected choice contributed to model quality.

Second, another option would have been to select a threshold maximizing the difference in means (between reused and un-reused macros), rather than the difference between above-threshold proportions. Evaluating this decision (below) showed that it had a negligible impact.

Measures and goals for prediction quality

Ten-fold cross-validation is the usual method used when training and evaluating a model on the same data set [72]. This approach seemed sensible for comparing the quality of the model with the quality of the model variants.

A variety of quality measures have been used in machine learning research. The approach used here of training a model on macro traits in order to predict reuse resembles the standard approach in software engineering defect prediction, which trains a model on module traits (static code traits or process data) in order to predict the presence of defects. The primary quality measures used in that literature are False Positive (FP) and True Positive (TP) rates (Table 16) [19][53].

Table 16. Quality measures commonly used in defect prediction literature

		Actual	
		Reused	***Un-reused***
Predicted	***Reused***	a	b
	Un-reused	c	d

$$TP = a/(a + c) \qquad FP = b/(b + d)$$

TP is the same as the recall measure used in information retrieval, indicating the fraction of interesting items (reused macros) that are successfully identified. FP indicates the fraction of uninteresting items (un-reused macros) that are erroneously identified. It is similar in purpose to the precision measure $a/(a+b)$ used in information retrieval to quantify prediction specificity, but FP is often preferred over precision in defect prediction. One reason for this preference is that FP is more stable than precision to small changes in the data when $b + d \gg a + c$ [69], which is often the case in defect prediction and certainly the case in web macro reuse.

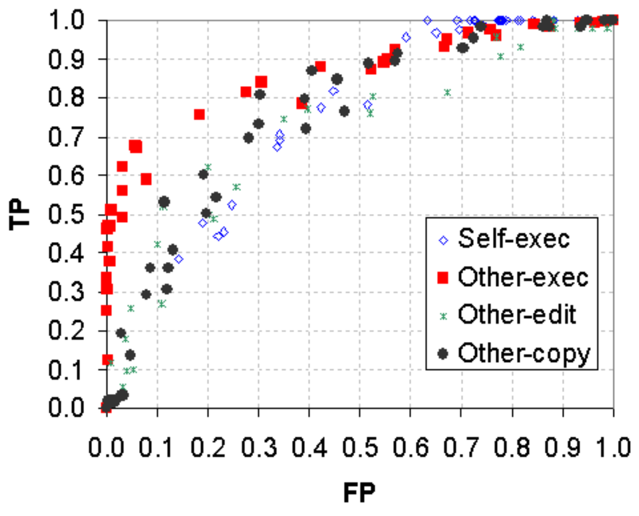
In terms of choosing target values of FP and TP, "Detectors learned in the domain of software engineering rarely yield high precision [low FP] detectors" [69]. For example, $FP \geq TP - 0.3$ for many defect prediction models [75]. Such high FP is usually acceptable because these models are generally used to identify items that should be brought to the attention of humans, who can then review the items. For applications like these, TP is somewhat more important than FP, since a person cannot review an item unless it is first brought to his attention. Menzies et al's review of software engineering papers that use machine learning [69] shows that at least in software engineering, TP is often around 0.7 or less. In fact even skilled software engineers achieve a recall of only 0.6 when they manually review modules for defects [116]. Meeting or exceeding these results calls for a model goal of $TP > 0.7$ and $FP < 0.4$.

Results: predictive power of reusability traits

For each reuse measure, 10-fold cross-validation at varying levels of α and β showed that TP reached the goal of 0.7 when FP ranged from 0.05-0.35, depending on the reuse measure (Figure 53). Conversely, TP was in the range 0.7-0.8 when FP reached 0.4. So for each measure, the model exceeded the goals.

For the *self-exec* reuse measure, FP is closest to 0.4 at $\alpha = 0.16$ and $\beta = 5$. For the other measures of reuse, FP is also closest to 0.4 at $\alpha = 0.16$ but with $\beta = 3$. Adjusting α and holding β constant, or vice versa, generated the full range of TP and FP.

Figure 53. Model quality at various parameter values



As noted above, the training algorithm could have used other heuristics for selecting the optimal threshold for each trait. To evaluate the algorithm's design decisions, I repeated the evaluation using variants of the model that selected each adjusted trait's optimal threshold τ_i based on difference-in-means or difference-in-entropy rather than difference-in-proportions (Table 17). The results confirmed the design decision, as difference-in-proportions yielded slightly higher quality than difference-in-means and substantially higher quality than difference-in-entropy.

Table 17. TP at FP $\approx$ 0.4, with variant methods for selecting each trait's threshold

	Selecting thresholds based on…		
	Proportions	Means	Entropy
Self-exec	0.78	0.78	0.62
Other-exec	0.79	0.79	0.40
Other-edit	0.77	0.70	0.50
Other-copy	0.72	0.70	0.62

Information content of reusability traits

The bottom line of these experiments is that the traits generated by the considerations of Section 8.1 contained sufficient information to fairly accurately predict several measures of web macro reuse. In fact, these predictions did not even need to rely on very many macro traits. In particular, Table 15 shows shaded cells where macro traits were active at $\alpha = 0.16$ (FP = 0.4). At this level, 17 traits were active and accounted for the model's predictiveness. Shaded cells generally corresponded to cases where a trait had shown strong statistical correspondence to reuse. The match is not perfect since the z-score used in statistical tests depends not only on differences in proportions, but also the absolute sizes of these proportions. This is a common phenomenon in statistics: a difference might be statistically significant yet not meaningful. The most useful traits were in the Mass appeal, Functional size / length, and Authorship categories, with more minor contributions from the Understandability and Flexibility categories.

Depending on the particular measure of reuse, as few as 5 traits were active or as many as 11, suggesting that a repository designer might be able to collect fewer than a dozen different traits in order to drive reusability-oriented search features.

8.3 Reusability-oriented search features in the topes repository

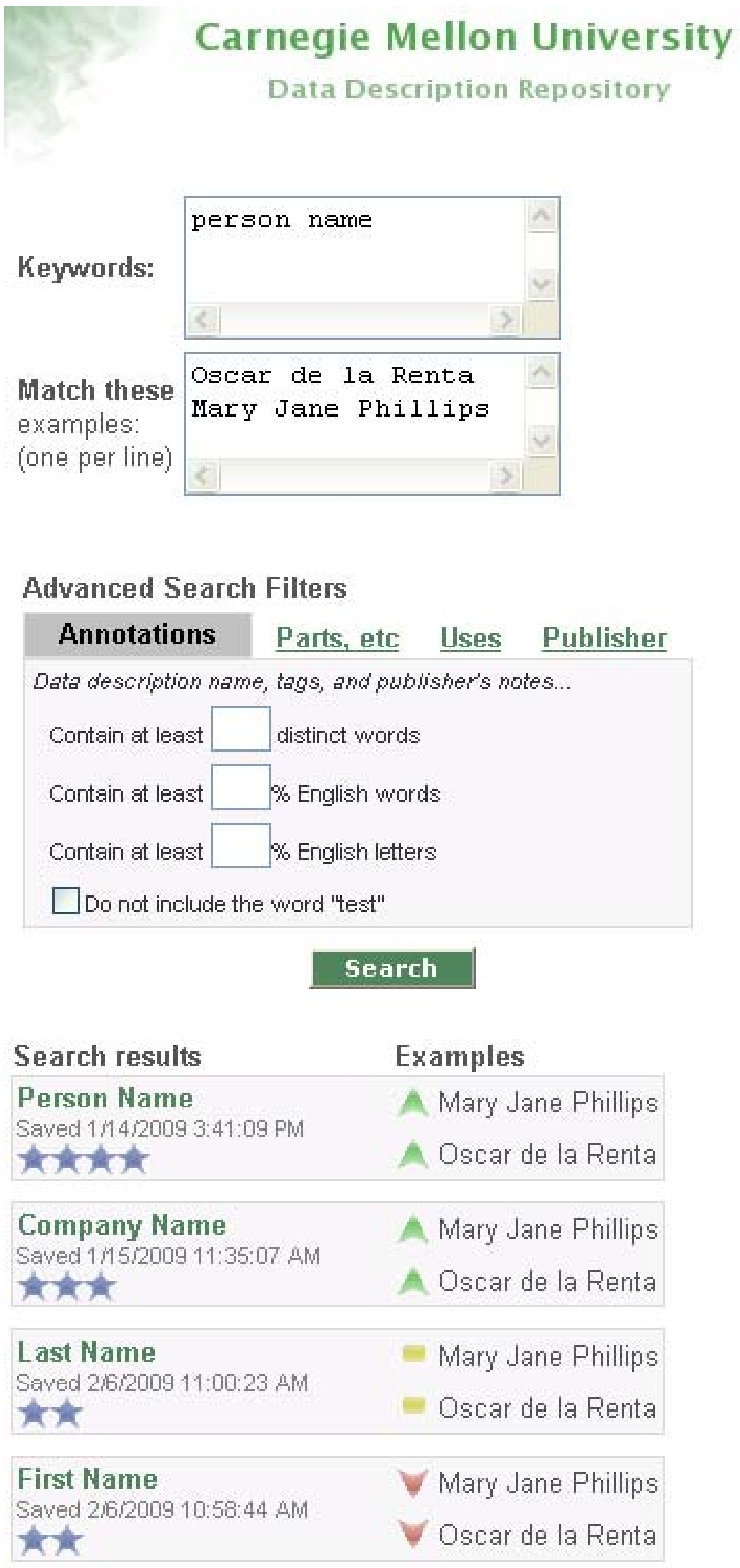
The empirical studies of end-user programmers described by Section 8.1 suggest that a wide variety of end-user code tends to be more reusable if it has certain traits. The CoScripter experiment described by Section 8.2 shows these traits can be used to predict whether macros will be reused. Together, all of these results suggest that a repository could effectively guide users to find reusable code.

Motivated by these considerations, the topes repository offers an "Advanced Search" feature that allows users to filter search results based on traits like those discussed by Sections 8.1 and 8.2. The intent of this design is simply to explore one way to incorporate the results of the empirical studies above into the design of a repository system. Usability studies would likely identify possibilities for moving forward further with the TDE repository's Advanced Search feature.

Advanced Search overview

After doing a basic search-by-match query, the user can activate the reusability-oriented filters by clicking on the "Refine results based on advanced search filters" link (Figure 39). The user interface shows a tabbed set of search filters (Figure 54). The user can enter values for one or more filters, then click "Search", which applies the filters to the search results.

Figure 54. Reusability-oriented filters in the search interface of the topes repository



Keywords: person name

Match these examples: (one per line)
Oscar de la Renta
Mary Jane Phillips

Advanced Search Filters

Data description name, tags, and publisher's notes...

Contain at least [ ] distinct words

Contain at least [ ] % English words

Contain at least [ ] % English letters

☐ Do not include the word "test"

Search results	Examples
Person Name Saved 1/14/2009 3:41:09 PM ★★★★	▲ Mary Jane Phillips ▲ Oscar de la Renta
Company Name Saved 1/15/2009 11:35:07 AM ★★★	▲ Mary Jane Phillips ▲ Oscar de la Renta
Last Name Saved 2/6/2009 11:00:23 AM ★★	▬ Mary Jane Phillips ▬ Oscar de la Renta
First Name Saved 2/6/2009 10:58:44 AM ★★	▼ Mary Jane Phillips ▼ Oscar de la Renta

Supported filters

Table 18 shows the list of search filters. Many of these are based on traits directly reflected in the code provided by the data description's author—where "code" is taken (as with web macros) to include comments and other annotations provided by the author. The traits reflected in code are based on the same general requirements for reuse mentioned by Section 8.1.1 and considered throughout this chapter: findability (with more reusable code generally having more mass appeal), flexibility (so that minimal modification is needed to fit into the reuse context), substantial functional size (so that minimal extension is needed to cover the reuse needs), and understandability (so that the user can make any necessary modifications or extensions). In addition, the topes repository includes several filters based on prior uses and authorship.

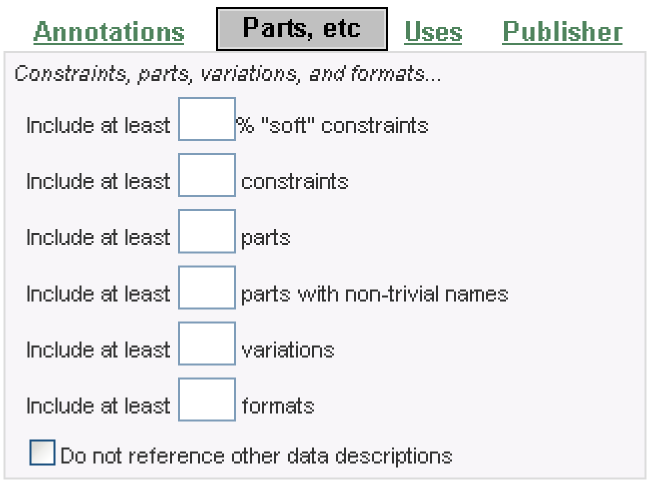
Table 18. Reusability-based filters in search interface of the topes repository

Tab on user interface	**Search filter**	**Reflected In**	(Related to)
Annotations	# distinct words	Code itself	Understandability
	% English words	Code itself	Understandability
	% English letters	Code itself	Understandability
	Do not include the word "test"	Code itself	Mass appeal
Parts, etc	% soft constraints	Code itself	Flexibility
	# constraints	Code itself	Functional size
	# parts	Code itself	Functional size
	# parts with non-trivial names	Code itself	Understandability
	# variations	Code itself	Functional size
	# formats	Code itself	Flexibility + functional size
	Have no references to other topes	Code itself	Flexibility
Uses	# prior downloads	Prior uses	
	Average rating	Prior uses	
	# times referenced by other topes	Prior uses	
Publisher	# previous topes published	Authorship	
	# previous topes rated	Authorship	
	# days since joined repository	Authorship	
	Email address has a certain domain	Authorship	
	Is from a certain geographic location	Authorship	
	Is a system administrator	Authorship	

The first tab, "Annotations", contains four search filters driven by the name, tags, and notes entered by the data description's author through Toped^{++} (Figure 54). (The name is entered on the main tab of Toped^{++}, shown in Figure 10, while the tags and notes are entered in the "Notes" tab of Toped^{++}.) These filters allow the user to restrict the result set to only include data descriptions that have a substantial amount of intelligible annotation. During the CoScripter experiment, macro reuse was related to several traits based on the "English-ness" of the annotations, which motivated the inclusion of analogous search filters on the "Annotations" tab of the topes repository's search filters. Conversely, the presence of the word "test" corresponded to lower macro reuse, which motivated a comparable topes repository filter.

The "Parts, etc." tab of the tope repository's advanced search is based on statistics concerning the number and characteristics of each data description's constraints, parts, variations, and formats (Figure 55). A high concentration of soft constraints (those that are not "always" or "never" true) might indicate higher flexibility, as soft constraints might enable a tope implementation to gracefully question but not reject unusual inputs that the original author did not anticipate. Adding parts and variations (and, consequently, formats) to a data description increases its functional size, raising the amount of time potentially saved by reusing it. In addition, the provision of more formats aids in flexibility by allowing the tope implementation to recognize a wider variety of strings. On the other hand, if a data description has dependencies on other data descriptions, then this can reduce flexibility, since the referring data description can be "broken" if the referenced data description changes.

Figure 55. Advanced search filters based on constraints, parts, variations, and formats.



Constraints, parts, variations, and formats...

Include at least ☐ % "soft" constraints

Include at least ☐ constraints

Include at least ☐ parts

Include at least ☐ parts with non-trivial names

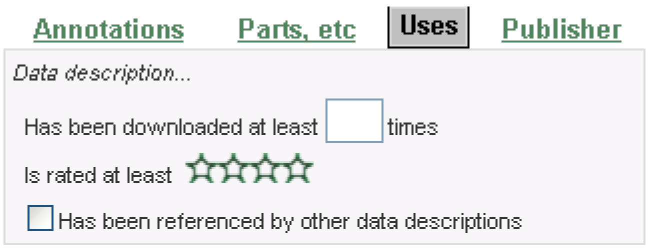
Include at least ☐ variations

Include at least ☐ formats

☐ Do not reference other data descriptions

The "Uses" tab includes three simple filters based on information about prior uses of data descriptions (Figure 56). The first filter, like the primary reusability-oriented feature of the CoScripter repository, is based on a count of how often each item is downloaded. The second filter is based on the average rating of data descriptions; these ratings are collected through a "Comments" form that appears at the bottom of each data description's "detail" page (scrolling down the inner frame of Figure 40). The third filter on the "Uses" tab is based on a count of how many other data descriptions in the repository refer to the data description, under the assumption that data descriptions are probably more reusable if they have already been successfully reused in multiple other data descriptions.

Figure 56. Advanced search filters based on prior uses of the data descriptions.



Data description...

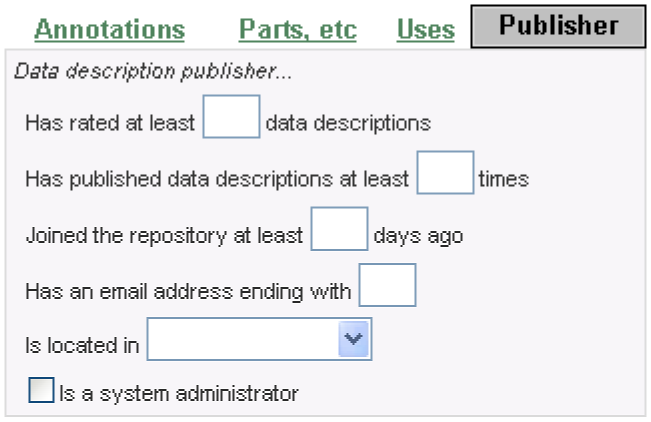
Has been downloaded at least [ ] times

Is rated at least ☆☆☆☆☆

☐ Has been referenced by other data descriptions

The final tab, "Publisher", is driven by information collected about the author of each data description (Figure 57). In the CoScripter experiment, several traits representing the author's prior experience were related to levels of macro reuse, motivating comparable traits in the topes repository. In addition, several macro traits indirectly corresponded to the date in which an author adopted CoScripter and also related to levels of reuse; these motivated a filter in the topes repository to focus search results on data descriptions that were created by people who created a repository account many days in the past. The next search filter is based on the email address of authors, which was motivated by the recognition that many topes are organization-specific. The penultimate search feature is motivated by the recognition that many topes are tied to geographical region, and it allows the user to filter search results based on what geographic region each data description author specified when creating a repository account; this piece of information must selected from a drop-down list showing all the world's continents, broken into several smaller regions such as North America, South America, Central Asia, and East Asia. Finally, since tutorial web macros created by CoScripter system administrators tended to be highly reused, the topes repository includes a filter for showing only data descriptions created by users flagged as repository administrators.

Figure 57. Advanced search filters based on information about authorship.



Data description publisher...

Has rated at least ___ data descriptions

Has published data descriptions at least ___ times

Joined the repository at least ___ days ago

Has an email address ending with ___

Is located in ___

☐ Is a system administrator

Possible future next steps

The filters shown here are not intended as a definitive final answer for how to provide reusability-oriented search features in the tope repository. Rather, they are intended to illustrate that it is straightforward to create search filters directly tied to the end-user code traits that have empirically corresponded to reuse in many studies, including the CoScripter experiment.

There is not yet a large multi-user tope repository for performing an experiment to directly evaluate how well these search features would help people to reuse topes. Nonetheless, many end-user repositories are quite successful despite providing a far less comprehensive suite of search features. This fact, combined with the empirical results described in the preceding sections, suggests that reusability-oriented search filters may provide a basis for helping end users to locate reusable tope implementations.

Future implementations of this approach might build on the basic user interface, shown above. One possibility would be to provide sliders for the fields that require numeric inputs, with notches in the sliders indicating suggested values; these values might be determined using an algorithm similar to that described in Section 8.2. Another possibility would be to track what data descriptions a user has downloaded and rated highly, then to train a machine learning algorithm to recommend other data descriptions that have reusability traits similar to those of the user's preferred data descriptions. Section 9.2.5 discusses other possible future opportunities to enhance the TDE topes repository.

Chapter 9. Conclusion

A series of empirical studies identified a common problem in applications for end users: these applications lack adequate support to help end users validate and reformat instances of data categories containing short, human-readable strings. In response to these user needs, this dissertation has presented a new kind of abstraction and supporting development environment that helps users to quickly and effectively validate and reformat data. This approach relies on a new abstraction called a "tope", which describes how to validate and reformat instances of one data category, such as phone numbers. Validation is non-binary, in that it can identify strings that are questionable and might be either valid or invalid.

Topes enable users to more quickly and effectively complete validation and reformatting tasks than they can with currently-practiced techniques. Moreover, because topes capture domain knowledge in a form that is application- and user-agnostic, they are highly reusable. Thus, topes not only help end users to succeed in their immediate tasks, but topes also can facilitate the successful completion of future tasks in a wide range of applications. These results are indicative of the potential benefits that can accrue from research aimed at helping end users to create and use models of data encountered in everyday life.

After Section 9.1 describes the contributions and claims supported by this work, Section 9.2 presents several opportunities for future work, including prospects for further extending and applying the topes model and the TDE. Section 9.3 concludes by discussing the implications of this thesis research, such as the desirability of end-user programming research whose results generalize beyond particular applications or programming tools.

9.1 Contributions and claims

A series of empirical studies have uncovered many tasks during which users need to validate or reformat short, human-readable strings. In most cases, these strings are valid if they are instances of a certain category, such as university names or corporate project numbers. Many such categories lack a formal specification, with the result that there often are questionable strings that look unusual but yet are valid. These categories are also often multi-format, in that strings of many data categories can be “equivalent” but written in multiple formats. Many of these categories are organization-specific, so developers of applications for end users cannot anticipate and support all of these data categories. In fact, most applications have virtually no support for automatically validating and reformatting strings. As a result, users have had to do validating and reformatting manually, which is tedious and error-prone. The finished data are sometimes questionable or even invalid, and they are often inconsistently formatted.

In response to the needs of end users, this research has produced the following six primary contributions:

- A new kind of abstraction, called a “tope”, providing operations for validating and reformatting short, human-readable strings in data categories, where those strings may be questionable and may appear in different formats
- Algorithms for generating tope implementations from data descriptions provided by users, and for executing tope implementations’ operations on data
- User interfaces and supporting code that enable users to create accurate data descriptions, then use the generated tope implementations to quickly and effectively validate and reformat strings
- A software architecture that effectively factors out application-specific concerns from application-independent validation and reformatting concerns
- A parallelizable “search-by-match” algorithm that searches through existing tope implementations to find those that match user-provided keywords and example strings
- A repository server that extends the search-by-match algorithm with features for filtering tope implementations based on traits that are generally useful for identifying reusable end-user code

The subsections below express three specific claims about these contributions, along with evidence supporting those claims. These results show that topes provide practical value right after they are created, and that they can be reused over and over to provide ongoing value in a wide range of applications and to multiple people.

Topes and the TDE are sufficiently expressive for creating useful, accurate rules for validating and reformatting a wide range of data categories commonly encountered by end users.

In the form of `isa` and `trf` functions, Topes provide functionality for validating and reformatting the short, human-readable strings that should be instances of data categories. By implementing topes and using them to validate and reformat a wide range of spreadsheet and web form data, I have shown that the TDE makes it possible to express useful validation and reformatting operations for several dozen of the most common kinds of string data encountered by users (Sections 5.2.1, 5.2.2, and 6.2.1).

In particular, the evaluation with spreadsheet data shows that through the TDE, topes can express validation that is three times as accurate as validation based on regular expressions (Section 5.2.2). This evaluation confirmed that topes were able to ferret out questionable strings, which yielded some of the benefit over regular expressions. Most of the improvements in accuracy came from the simplicity of packing many formats into a tope with the TDE.

In short, topes are capable of expressing useful validation and reformatting operations for short, human-readable strings. Each tope corresponds to a particular data category. Each tope can validate strings on a non-binary scale if so required by its data category. These results show that the topes model and TDE meet the *expressiveness* requirements identified in Chapter 1 (summarized in Table 6).

By using the TDE to implement and apply topes, end users can validate and reformat strings more quickly and effectively than they can with currently-practiced techniques.

Two experiments have confirmed that the TDE enables users to validate and reformat data more quickly and effectively than is possible with currently-practiced techniques. First, a between-subject experiment has examined how well the TDE helps end users to validate strings (Section 5.2.3). With the TDE, users finished their tasks twice as fast and found three times as many invalid strings as with the comparison system, Lapis [70]. These results also compare favorably with statistics provided by an earlier study involving another system, SWYN, which was based on regular expressions [16]. Second, a within-subjects experiment has shown that users can implement and use topes to reformat spreadsheet cells, and that this process is so fast that the cost of implementing a tope is "paid off" after only 47 spreadsheet cells—the alternative being to do the work manually, which is the only real option for users right now (Section 6.2.2). Both experiments included a user satisfaction survey, which confirmed that users are extremely satisfied with the topes-based approach and eager to see it deployed in commercial applications for end users.

Since topes can be created by people inside organizations, and they enable people to more quickly and effectively validate and reformat strings than is possible with currently-practiced techniques, these results show that the topes model and TDE meet the *usability* requirements identified in Chapter 1 (summarized in Table 6). Thus, topes are not only theoretically expressive enough to be useful, but they are usable enough to provide practical value to real users.

Tope implementations are reusable across applications and by different people.

For a tope implementation to be reusable in different applications by different people, it must be callable from those applications, it must be findable, and it must be capable of robustly validating and reformatting data regardless of which application provided the data.

To support cross-application reuse of tope implementations, the TDE provides a programmatic API called from applications via add-ins. The API effectively acts as a partition between application-specific concerns handled in the add-ins and application-independent concerns handled in the TDE. The feasibility of cross-application reuse has been demonstrated by including add-ins in the TDE for many different applications (Section 7.1.1).

In order to help find an existing tope implementation, the TDE provides a heuristic-based "search-by-match" algorithm that finds topes based on how well they match keywords and example strings. When tested on tope implementations for the most common few dozen kinds of data, the algorithm demonstrated a recall of over 80% and query time of under 1 second, which is sufficient for implementing user interface features that search through the collections of tope implementations that users are likely to have on their computers (Section 7.2.2). An analysis of this algorithm shows that it is highly parallelizable and suitable for searching through repositories of many more tope implementations (Section 7.2.2). A prototype repository system extends the basic search-by-match algorithm with additional search features aimed at helping users to filter search results based on whether tope implementations present certain traits. A variety of empirical studies show that traits like these tend to be useful for identifying reusable end-user code in general (Sections 8.1 and 8.2).

Finally, I tested whether tope implementations created for several dozen categories of spreadsheet data would be equally accurate at validating and reformatting data from web applications (Section 7.2.1). This evaluation showed that the tope implementations actually were more accurate on the web application data than on the spreadsheet data, largely because the web form data corpus demonstrated a narrower variety of formats for each category. In addition, calling reformatting functions proved to be an effective means of putting web application data into a consistent format and ultimately identifying duplicate values (Section 6.2.1).

In short, topes are application-agnostic abstractions whose implementations can be published, found, shared, combined, and customized. These results show that the topes model and TDE meet the *reusability* requirements identified in Chapter 1 (summarized in Table 6). Thus, topes not provide practical value right after they are created, but they can be reused over and over to provide ongoing value in a wide range of applications and to multiple people.

9.2 Future opportunities

This section concludes this dissertation by exploring a few opportunities for incremental improvements on topes and the TDE, then by moving on to more significant avenues for extending, generalizing, and applying the results of this thesis research to other forms of end-user programming.

9.2.1 Generalizing the combining of constraints in validation functions

The TDE generates `isa` functions that check each input string against constraints specified in a user-provided data description. Each constraint carries a penalty between 0 and 1, and if a string violates more than one penalty, then the `isa` function multiplies these penalties to generate a validity score. Multiplication is just one conceivable way of combining penalties, but it is worth considering what other approaches might be tractable and useful.

Multiplication is a reasonable starting point for combining penalties because each constraint's penalty is an estimate of the probability that the constraint will be violated by valid strings, so the product of the penalties yields an estimate of the joint probability that a valid string will violate both constraints. Since this computation yields a number in the range 0 and 1, it is an acceptable fuzzy set membership function and therefore a suitable way to implement `isa` scores.

Nonetheless, there are two ways in which the resulting `isa` scores could be adjusted so that they are more accurate estimates of the joint probability function.

First, each constraint's penalty is computed based on a simple interpretation of an adverb of frequency selected by the user for the constraint. These adverbs of frequency were selected because there is surprisingly little variance (on the order of a few percent) in the probabilities that people assign to these words. Yet there is still a non-zero variance in this mapping, and there is always the potential for people to select an incorrect adverb. One approach that might improve on the current situation would be for the TDE to dynamically update probability estimates by tracking how often each constraint is actually violated by valid strings, then using that information to update the base figure specified by the user. Bayesian statistics would be a natural and easy way to combine these pieces of information, though empirical investigation would be required to determine what weighting factors (if any) would be appropriate. This determination would not be trivial, since the resulting factors would either need to be universally applicable, or mechanisms would be needed to determine the appropriate factors for different situations.

Second, the improvement described above could be taken one step further: the TDE could track how often different combinations of constraint violations occur in valid data. Since there are 2^n combinations of n constraints, it would probably not be sensible to track every single combination's probability of "co-violation". Instead, perhaps the TDE could select only certain constraints for co-violation tracking based on information theoretic principles. For example, constraints that tend to be violated often (on their own) are probably also relatively likely to be violated at the same time as other constraints; thus, the TDE might select constraints for co-violation tracking if their adverbs of frequency (or Bayesian posterior) indicated a high likelihood of violation. With statistics of co-violations in hand, the TDE could then directly compute the actual joint probability that constraints might be violated by valid strings.

While the two possible improvements described above aim to tighten the correspondence between `isa` scores and joint probabilities, there are also other entirely different approaches for combining constraint penalties into `isa` scores. Most of these approaches do not aim to produce an `isa` function that corresponds to a true probability. Instead, they might offer alternative ways to tune the sensitivity of the `isa` function to multiple constraint violations. Section 3.4 pointed out one such alternate approach based on using arithmetic minimums of several scores. A second approach would be for the different constraints to "vote", and to take the arithmetic average of their results.

It would certainly be possible to invent many other alternate calculi for reasoning about constraint violations, but it is likely that each form of reasoning has its limitations. In this regard, the problem of identifying invalid strings resembles program verification problems encountered by other researchers in other domains, where no single approach was universally superior to the others. For example, researchers have provided algorithms for detecting probable spreadsheet formula errors, and several calculi have been proposed for combining the results of these algorithms into what might be called a unified spreadsheet "formula error detector" [54]. But preliminary empirical investigation shows that each approach has strengths and weaknesses and might be preferred in some situations but not in others [54]. Similarly, while the multiplication of probabilities in the TDE prototype serves as an effective start for implementing `isa` functions, it might be desirable to develop alternative approaches for implementing `isa` functions and then to empirically evaluate the strengths and weaknesses of these approaches.

9.2.2 Assuring consistency between topes and tope implementations

As explained by Section 3.5, a tope does not provide formal mechanisms to prevent users from creating a tope implementation that is inconsistent with its tope. The reason is that it seemed unlikely that users would be capable of or interested in writing a formal specification of each tope. As a result, this thesis research has relied on providing testing features and algorithms for detecting certain kinds of errors, with the intent of helping users understand whether the tope implementation matches the intended tope. These features include the testing tab of Toped^{++} (Section 4.3), algorithms for automatically detecting if ambiguity might have caused an incorrect reformatting (Section 6.1.2), and algorithms for automatically detecting inconsistencies between parts in different variations of a data description (Section 7.1.3). Personally, I have found the testing tab to be very useful almost every time that I implement a tope, and I observed during the empirical studies that users also extensively relied on this feature. The other features for automatically detecting errors are only useful once in a while.

Perhaps the features for automatically detecting errors could be made more useful if they were more aggressive about ferreting out potential errors in a tope implementation. For example, it might be useful to augment them with an algorithm that takes the strings provided in the testing tab and "runs them around" the tope graph by successively calling `trf` functions and calling `isa` functions at each step of the way. The algorithm could signal potential errors by identifying situations where `trf` functions turn valid strings into invalid or questionable strings.

With each new design-time static analysis aimed at detecting certain classes of errors in tope implementations, it would also be desirable to provide users with a way to quickly fix those errors. For example, consider the existing algorithms for automatically detecting inconsistencies between parts in different variations of a data description. Toped^{++} provides a simple "Find/Replace" dialog window so that it is very easy to fix these inconsistencies.

Likewise, with each new runtime dynamic analysis aimed at detecting when a tope implementation is producing invalid outputs, it would be ideal to provide users with a way to override or otherwise deal with those errors. For example, consider the existing algorithms for automatically detecting if ambiguity might have caused an incorrect reformatting. When strings are identified by this algorithm as potentially incorrect reformatting outputs, they are flagged so that the user can review the strings and fix them if needed.

Design-time and runtime algorithms aimed at detecting errors might still be valuable even if the user is never provided with any way to deal with those errors. One reason is that the error-detection algorithms could be incorporated into the repository to provide another measure of reusability. Another reason is that the algorithms could be used for research purposes to analyze tope implementations created by end users, in order to provide additional measures of correctness beyond the F_1 measure used by this dissertation.

9.2.3 Composition and decomposition of topes

A tope's interface exposes `isa` and `trf` functions that each accept a string parameter. Consequently, with a single function call, it is only possible to validate or reformat a single string. It is not possible to issue a function call that validates or reformats several strings of different data categories. This limits topes' ability to detect certain kinds of errors.

For example, a spreadsheet might split a mailing address into several cells. One cell might contain the street address, another might contain the city, another might contain the state, and another might contain the zip code (as illustrated in Figure 15). The user could assign a different tope to each of these cells with the current tope model and Excel add-in. The TDE could then validate and reformat each of those cells. And, in fact, the evaluations described by Sections 5.2 and 6.2 show that topes and the TDE do enable users to validate and reformat data more quickly and effectively than is possible with currently-practiced techniques.

But it might possible to go even further, to address situations where the validity of a string depends on nearby strings. Returning to the mailing address example, the street address string should correspond to a real building located in a real city referenced by the city string. The city, in turn, should refer to a real city located in the specified state. Together, physical location together identified by the street address, city, and state should be geographically positioned within the real region referenced by the zip code string. (Incidentally, zip codes can cross state boundaries.) Validating each of the strings with a separate tope, as in the current model, fails to capture these semantics.

One way to address this limitation would be for the add-in to let the user concatenate the strings (with some sort of separators) and feed the resulting string into a tope specifically designed to validate the entire mailing address. Actually modeling such a data category would be well beyond the capabilities of the basic data description editor. Theoretically, the necessary validation rules could be expressed with JavaScript "linking constraints" in Toped^{++}. But in reality, it might be more efficient if the data description could instead act as a stub to call a web service validates the entire string. For example, the data description could pass the string to maps.google.com and see if the server returns the HTML for a map.

One problem with this approach is that it needlessly requires parsing the concatenated string. A more efficient approach would be to create a sort of "record-tope" with validation and reformatting functions that accept more than one string. That is, whereas a tope's operations accept a *single* string (that might be a concatenation of multiple meaningful sub-strings), a record-tope's operations would accept *multiple* strings. Each parameter would need to match a certain tope (which could be a multi-format tope of the kind described in this dissertation, or it could in turn be a record-tope). For example, a validation function record-tope for mailing addresses might accept an address line, a city, a state, and a zip code. It would not stipulate that these parameters should match any particular format of the corresponding tope. For example, the first actual parameter could be an address line written as "12 Main Street" or "12 MAIN ST." or "12 Main Str." After all, the record-tope could always call referenced topes to put actual parameters into whatever format is preferred internally by the record-tope.

Another useful extension of the topes model might be for each tope to provide functions for *decomposing* strings. The TDE actually does provide functions for parsing a string and retrieving the parts of the string that are named by the data description. For example, an address line string could be parsed with the TDE to retrieve the street number, predirectional, street name, postdirectional, and street type (or whatever other parts happened to be named by the address line data description). This has proven useful for writing unit tests to help me ensure that the TDE is performing properly. Actually exposing this functionality through the topes interface might be useful for helping users to automate tasks that involve decomposing strings. Further research will be required to precisely characterize situations where users must perform such tasks manually, as well as to empirically evaluate what user interfaces might enable users to quickly and correctly invoke string decomposition functions.

9.2.4 Integrating topes into other applications and programming tools

As described by Section 7.1.1, topes have already proven useful for validating and reformatting strings in web forms, web macros, and spreadsheets. Other potential applications abound, including the potential for integrating topes into the operating system, textual programming languages, and various additional end-user programming environments.

Integrating topes with the operating system

When a user needs to transfer data from one application to another, the data might need an intervening transformation. In such situations, integration with topes at the operating system level might help to facilitate inter-application transfers.

For example, Section 2.3 mentioned that administrative assistants often need to copy data from spreadsheets into web forms when creating expense reports. Sometimes, the data in the spreadsheet is not in the format required by the web form, which has forced users to manually reformat data after copying and pasting it. This thesis research has provided two ways to address this problem: the user can now either use a tope in the spreadsheet environment to automatically reformat (and then copy-paste it into the web form), or the user can simply copy-paste the string in its old format into the web form and let the web form automatically reformat the string as needed (assuming that the web form's programmer has attached a tope to the web form input field).

But what if the user has a spreadsheet editor that does not yet have a TDE add-in, and the web form field does not have an associated tope for reformatting the string? Then the user is forced to manually reformat the string, returning to the tedious and error-prone process that this research aimed to avoid.

One way to address this problem would be to integrate topes into the operating system clipboard in order to support a form of "smart copy-and-paste". After the user copies a string from the source application, he could press a special key combination (Ctrl-Shift-F11?) signaling to the operating system that reformatting is needed. The operating system could bring up a list of recommended topes and their available formats by calling on the algorithm described by Section 7.1.2. After the user selects or creates a new tope and format, the operating system would call the specified tope to reformat the clipboard's string. This new string would be stored in the clipboard so that the user could proceed with a paste operation as usual.

Even better, since users often need to copy and paste the same kinds of data over and over (as when filling out many expense reports from several co-workers), the operating system could track a list of recently used topes. Then, then when the user needs to reformat a string using a recently-used tope, he could press a key combination (Ctrl-Shift-**F12**?) instructing the operating system to put the clipboard string into the most recently used tope for that kind of data. The operating system would take the clipboard string, retrieve the list of recommended topes, select the recommended tope that was used most recently, reformat the string to the format that had been selected the last time that the tope was used, and then put the new string onto the clipboard. This would allow the user reformat a string in a single keystroke, without having to repeatedly browse through the list of recommended topes and their formats.

A further extension would be to integrate this basic smart copy-and-paste with the "record-tope" model outlined by Section 9.2.3. Users would be able to define a record-tope, copy strings into a clipboard, use the record-tope to reformat the string and to "burst" the string's parts into sub-strings, and then simultaneously perform multiple paste operations to transfer the sub-strings into multiple text widgets. This would effectively generalize the capabilities of the Citrine tool, which can copy, decompose, and paste certain kinds of data into *multiple text widgets* at the same time [118]. For example, a user could copy a mailing address from an email message and then paste it into multiple web form fields at the same time. At present, Citrine only supports a small, restricted set of certain data categories (mailing addresses, research paper citations, personal contact data such as phone numbers, and event-related data such as time and date); another limitation is that Citrine cannot perform intervening transformations. Integrating record-topes with the smart copy-and-paste functionality described above would eliminate these two restrictions, since users could define new record-topes for custom data categories, and they could invoke the reformatting rules of the topes referenced by a record-tope.

Integrating topes with traditional textual programming languages

As described by Section 2.7, professional programmers often need to create "domain value" classes in order to work around the limitations of the object-oriented type systems. For example, the programmer might create an "interest rate" class for checking whether a value like "2.5%" is a valid interest rate. Right now, programmers would need to manually write the code for implementing such a class. This code might parse the string "2.5%" to strip the "%" symbol, convert the "2.5" into a floating-point number, check whether 2.5 falls within an acceptable range, and finally return true or false.

Topes offer the possibility quickly implementing such classes. A programmer might define a data description for interest rates. An extension of the TDE (perhaps an add-in to Eclipse) might automatically generate a Java class with a method that accepts a string, then checks the string against a tope generated from the data description. The method would return true if and only the string had an `isa` score of 1.

Moreover, topes offer the possibility of going beyond traditional domain value classes. A class's validation method could actually return the `isa` score (rather than just converting it to a Boolean), thereby making it possible for whatever code calls the validation method to ask users for help with questionable values. Furthermore, the domain value class could expose all of the tope's reformatting functionality.

Integrating topes with various additional end-user programming environments

Preliminary steps have already been taken toward integrating topes with other end-user programming environments. For example, I have developed a very minimalist proof-of-concept add-in for Microsoft SQL Server (described in [100]) for validating strings in database columns. In addition, I have experimented with integrating topes into Adobe Dreamweaver, a popular end-user tool for designing web pages and web forms. Two companies, RedRover and LogicBlox, have identified opportunities for integrating topes into spreadsheet auditing and decision support software, respectively. Other researchers have suggested integrating topes with Matlab, Forms/3, and analysis frameworks for identifying dimension-unit errors. Additional opportunities will probably arise if topes become widely popular and come to the attention of many software development companies.

9.2.5 Repository enhancements addressing tope proliferation

If topes do become widely successful, they might begin to "proliferate in the wild". What are the potential problems associated with proliferation, and how might these problems be countered? These problems can be grouped into three logical categories: problems searching repositories for topes, problems with deciding whether to use one of the search results or to create a new tope, and problems with managing topes after they are on the user's computer.

Searching through large repositories for topes

While the evaluation in Section 7.2 showed that the tope recommendation algorithm can search through a few hundred topes within a second, and it showed that the algorithm could be parallelized in a way that remains fast even as the number of topes grows further, this evaluation did not address whether the search results would remain *accurate* for very large tope repositories. Thus, if more topes eventually become available, it will be necessary to determine how well repositories are able to find topes for the data that users have at hand.

The scalability of the search algorithm actually might not turn out to be a significant problem, however, since so many data categories are organization-specific: users might naturally publish most topes to organization-specific repositories. This would effectively partition the total set of topes into smaller, more searchable sets.

Helping users to decide whether to customize topes in the search results or to create from scratch

On the other hand, the organization-specific nature of so many data categories might also work against the usability of the repositories, since users might frequently find that the available topes are *not quite* what they were looking for. For example, the available topes might lack certain organization-specific formats or organization-specific constraints. Conversely, the available topes might have unusual format or constraints that are inappropriate for a particular setting.

In such situations, users have a choice of downloading an available tope implementation and customize it, or starting from scratch by creating their own. The TDE provides functionality for both strategies (as discussed by Chapters 4 through 7). The question is, how can the TDE (particularly the repository) help the user to decide which strategy to apply?

The current repository prototype offers some assistance with this: users can provide examples of strings that they would like a tope to match, and the repository will display visual indicators showing whether the available topes matched those examples (Figure 16). Going one step further, the repository could actually show the usual human-readable, targeted error messages actually explaining why strings do not match the available topes, thereby calling out specific constraints that the user might be able to remove or modify in order to tune topes to the current need. In addition, if the repository allowed users to specify strings that should *not* be accepted by a tope, then the TDE could propose constraints that should be added to existing data descriptions; this would require incorporating fairly standard machine learning techniques (such as entropy maximization [72]) into the tope inference algorithm.

Helping users to manage customized versions of topes

If a user does decide to download and customize a tope implementation, then a new problem arises: what to do if the original tope implementation changes? Should the user's customized version also change? Or should the user's customized version stay the same? Actually, some of the same considerations also apply even if the user never accesses any repository at all, since he might maintain many different versions of topes on his own computer, for use in specialized situations. For example, the user might create a project number tope for validating company data, but there might be a certain situation (monthly reports?) when only certain project numbers should be allowed, prompting the user to create a specialized project number data description that has an extra constraint.

One unified approach for addressing these considerations would be for the TDE to track the starting source of every data description, and then to periodically check whether a new version becomes available at that source. For example, if a certain data description A is downloaded from a repository, and the user customizes it to produce data description B, then the TDE would record the fact that B's source is A. If the user then further customized B to produce data description C, then the source of C would be B. Thanks to the REST-based protocol implementation of the repository (Section 7.1.3) all such source pointers can be represented as URLs. (The source of B would look something like "http://repositoryserver.domain.com/path/webapp.aspx?URL=tope-guid", while the source of C would look like "file://c:/documents and settings/user/local settings/Application Data/topesv4/tope-guid.xml").

The TDE could poll remote URLs for changes to remote sources' implementations, and it could listen for file system events to detect changes to local sources' implementations. At a reasonable, configurable level of frequency, it could then offer to download remote implementations (with a standard Windows system tray popup like, "You have available updates. Upgrade now?").

In order to help users make a reasoned decision about whether to accept remote updates, the TDE could log strings that have been tested recently with the current data description, and it could show a summary of strings that the new data description would validate or reformat differently. This essentially would represent an end-user-friendly form of regression testing. Moreover, if the new tope implementation would treat those strings differently than the current version does, then the TDE could identify specific constraints in the old or the new implementations that are causing the different behavior. Thus, the TDE could spell out exactly what has changed in the new version (and, perhaps, offer users the ability to accept or reject changes at a fine level of granularity).

Finally, if the user decides to accept an updated source implementation, the TDE could determine whether the user wants to propagate any changes to customized versions based on the changed source (regardless of whether the new source is remote or local). Internally, the TDE actually already provides a starting point for helping users to make this decision. Specifically, the TDE currently tags every part and constraint in every data description with a GUID; thus, the TDE could easily compute a "diff" to determine additions, modifications, and deletions made by the user during customization. To help the user decide whether to propagate changes to customized topes, the TDE could perform a "diff merge" of the new source with the user's customizations, and then use the regression testing approach to check whether the tope from the resulting merged data description would perform the same as the old customized tope. It could then spell out exactly what would the effects would be if the user chose to propagate changes to the customized version (and, perhaps, offer users the ability to propagate some changes but not others).

9.2.6 Beyond topes: helping users to create, share, and combine code

When considered as a general approach for helping users to accomplish their tasks, it is clear that end-user programming is extremely successful. One measure of success is the large diversity of end-user programming tools such as Excel and Matlab that have been commercially profitable for many years. Another measure of success is the sheer number of people who create programs with these applications—perhaps as many as 100 million worldwide [12]. Still a third measure of success is how *integrated* these applications have become with users' lives. For example, despite the non-trivial potential for errors in spreadsheets and web macros, many employees rely on these end-user programs in order to accomplish everyday work tasks [21][60][85].

However, end-user programmers run into trouble when trying to connect one piece of user-created data or code to another piece of user-created data or code. For instance, Section 2.3 mentioned that administrative assistants often need to copy data from a spreadsheet into a web form, but they had no tool that could automate this data transfer (including intervening reformatting). In this case, the administrative assistants needed to copy data from a spreadsheet created by another user into a web form created by professional programmers. Nonetheless, the administrative assistants would have had just as much trouble automating the transfer of data from a user-generated spreadsheet into a user-generated web form.

In fact, the administrative assistants might have had even *more* trouble automatically transferring data into a user-generated web form. One reason is that web forms produced by end-user programmers might be more likely to contain JavaScript bugs than web forms produced by professional programmers (potentially forcing the administrative assistant to put extra effort into fail-safe, error-resistant code). Also, web forms produced by end-user programmers might be more likely to be located on obscure or not-easily-discovered web pages, compared to web forms produced by the organization's professional information services staff (potentially forcing the administrative assistant to hunt for the web form's location before being able to automate the data transfer). Of course, this list of potential problems is not exhaustive, and there are probably many other problems that uses would encounter when trying to automate tasks like the ones performed by administrative assistants in the contextual inquiry.

To further illustrate with a more complex although hypothetical example, end users currently have no easy way to automatically read bank borrower names from another user's spreadsheet, to look up corresponding loan information from a database created by another co-worker, to compute each borrower's monthly interest payments with a script created by still another co-worker, and to display the results beside each borrower name in the original spreadsheet.

Automating tasks like these will be impossible until end-user programmers can easily create, share and combine code to produce a new program. The results from this thesis research might help researchers to overcome three key stumbling blocks that stand in the way of achieving these objectives.

First, end-user programmers need a reliable way of finding existing code to reuse and combine. The topes repository has offered a start in this direction, as it has provided a venue for understanding the traits of reusable end-user code in general, as well as for exploring one point in the design space for how those traits could be integrated into a search engine. Helping users to share and combine code could involve generalizing and further validating the notion of identifying reusable code based on these traits. In addition, it would be desirable to use these traits for other system features in addition to search interfaces. For example, if an end-user programmer uploads new code, the repository could check whether the code appears to be reusable and could issue suggestions for how the end-user programmer might improve the code. (The repository could check the code for other quality attributes other than reusability, such as checking for common performance problems or potential buffer overrun exploits.) If effective, such a feature could lead to a repository containing more useful code, which probably would indirectly make it easier for end-user programmers to find reusable code in the repository.

Second, after identifying useful existing code, end-user programmers need a reliable way to integrate code, which often requires writing code that performs intervening transformation operations. Topes' `trf` functions provide a beginning for implementing these operations, but further work will be required to fully automate data exchange between pieces of end-user code. Specifically, in this thesis research, transformations were only called under human oversight. If a string happened to be invalid, or if ambiguity could cause improper reformatting, an add-in can ask a user for help. But when automatically exchanging data between pieces of code, no human will be in the loop. In that case, it would be necessary to develop alternative approaches for automatically preventing or coping with problems.

Finally, there has been little work aimed at helping end users to *produce* code that other people can understand and reuse. For example, while virtually every empirical study mentioned by Section 8.1.1 emphasized the importance of code comments in promoting understandability, the same studies also showed that end-user programmers rarely take time to embed comments in code. End users lack the time to make significant up-front investments in understandability, which in turn hampers reusability by peers. As a step toward improving this situation, perhaps annotating spreadsheet cells, text input fields and other widgets with references to topes would help to make their programs more understandable (just as labeling variables with types is claimed to help improve the readability of professional programmers' code [87]). In order to reduce the time required for users to make these annotations, add-ins might recommend tope annotations by taking advantage of the TDE's search-by-match algorithm.

9.3 Conclusions

This research has investigated an approach for helping users to automate activities that they currently must perform manually. This approach is founded on the notion of bringing the abstractions supported by end-user applications into closer alignment with the data involved in the problem domain. For decades, researchers of human-computer interaction have warned that a poor alignment between constructs in computer systems and constructs in the problem domain can interfere with humans' ability to use those systems [37][81]. This research shows that improving this alignment is an effective way to help users to automate validation and reformatting of strings.

An immediate implication of this result is that researchers might be able to help end users with other programming tasks by finding additional situations in which programmatic abstractions have a poor match to the problem domain. In response, researchers could design new kinds of abstractions that are more closely matched to the problem domain. By instantiating and applying such abstractions, users might be able to more quickly and effectively complete everyday tasks.

Application-agnosticism as high-leverage research

Moreover, as exemplified by this thesis research, a possible consequence of closely matching programming abstractions to the problem domain is that the resulting abstractions can be *application-agnostic*. The direct benefit of this is that when a user creates an abstraction, he can then reuse it in many applications and possibly share it with other end users. This in turn enables users to build up a toolkit that they "leverage" in order to more quickly and effectively complete future tasks, in addition to the task that initially prompted the instantiation of an abstraction.

A second, even more significant benefit of research aimed at inventing application-agnostic abstractions is that the results are highly *generalizable*, as they can be immediately leveraged outside of one particular end-user programming application. This topes research has demonstrated this benefit, and much of topes' future research opportunities (Section 9.2) promise benefits only because topes are application-agnostic. Another initiative that has demonstrated similar benefits is the Open Minds Common Sense project, whose ConceptNet application-agnostic model captures common-sense relationships among entities (e.g.: dogs have teeth); researchers have tapped this model in order to provide powerful new features in nearly a dozen different user applications [61]. Similarly, Apple Data Detectors can be activated by any application on MacOSX simply by instantiating a text box widget with the new functionality for calling detectors and displaying the context-menu of available scripts [79], and Lapis patterns can be used to select text in HTML, Java, or ordinary text [70].

Yet, to date, a great deal of end-user programming research has instead focused on producing non-application-agnostic systems, algorithms, or programming tools, which can have the effect of "locking away" research contributions inside of particular applications. This can lead to multiple research teams producing fairly similar results that mainly differ only in their target application or programming environment. For example, Forms/3 can infer constraints for validating numeric data in spreadsheets [25], while Cues can infer constraints for validating numeric data in web service data [93]. Could these two constraint-inference models be generalized into application-agnostic models?

Would doing this make it possible to unify the models, extend the unified application-agnostic model, and apply it in other user applications where numeric data appear (such as Access databases and web pages)? On their own, each of these two research projects makes significant contributions and offers valuable insights, but what additional benefit might be gained by decoupling them from specific applications and then combining them? Other examples of contributions potentially worthy of "application-agnoticizing" include the file manipulation interface of Pursuit [73], the lightweight semantic labeling approach of HTML microformats [47], the set-selection abstractions of Hands [84], and the Potter's Wheel system for performing data cleaning on tuples [91]. Section 9.2.3 identified a related kind of data, which essentially consists of records of topes, that also might prove to be a worthy candidate for an application-agnostic abstraction.

Four keys to topes' application-agnosticism

One of the keys to topes' success as application-agnostic abstractions is that they are models of the users' "real world" problem domain rather than models of a particular solution domain. Topes are not models of strings in spreadsheets, nor models of strings in web forms. They are models of strings in data categories that belong to societies or organizations. The implication is that researchers might find it more straightforward to design application-agnostic abstractions if those abstractions are tied to phenomena in the real world (rather than one particular application).

Another key to topes' success as application-agnostic abstractions is the model's support for incremental modeling of the problem domain. Since so many data categories are organization-specific, it would have been impossible (or at least intolerably slow and costly) to design a single perfect tope ontology of the world. Consequently, the topes model is designed so that people can grow a toolkit incrementally by implementing and using topes as needed. Moreover, users can incrementally refine tope implementations over time, if so desired. The implication is that incrementalism might be a desirable feature to consider providing in other application-agnostic abstractions.

A third key to tope's success as application-agnostic abstractions is their support for semi-automated creation and use of domain models. This frees tope creators from the need to make perfectly precise topes. For example, as described in Chapter 2, one reason that web developers omit validation for web form inputs like person names is because they are worried that the validation will prevent people from entering unusual but valid inputs. With topes, this false choice between intended-to-be-perfect validation and no validation at all is removed: the programmer can specify validation that keeps out definitely invalid inputs but which allows users to override warning messages for questionable inputs. In addition, transformation rules only need to make a "best effort" and are allowed to produce invalid outputs if appropriate (which the validation rules can detect for double-checking). Thus, programmers can add new topes to their toolkit even for data categories that cannot always be precisely modeled. The implication is that semi-automation might be a desirable feature to consider providing in other application-agnostic abstractions.

Finally, perhaps the most significant key to topes' success is that they target a shallow level of semantic granularity appropriate for the problem domain at hand. In particular, topes are an entirely new kind of abstraction at a level of semantic depth that has not been thoroughly explored. Regular expressions recognize strings based on syntax, but they do not require strings to have any particular meaning. While tope `isa` functions recognize strings based on syntax, they are part of an abstraction intended to capture membership in a data category. This shallow semantics is deepened slightly through `trf` functions, which capture the equivalence of strings, usually in the sense of referring to the same real-world named entity (such as a person or a mailing address).

Thus, topes are a shallow-semantics abstraction describing one data category. They capture a level of semantics that is appropriate to their particular kind of data from the problem domain. There are other modeling approaches that allow much richer abstractions capturing deeper semantics. For example, Cyc and ConceptNet allow for relations among named entities [55][61]. Object-oriented systems generally allow for arbitrary operations on objects, polymorphism, and often even more exotic semantic constructs. These alternate approaches appropriately capture more semantics than topes do, since they attempt to model a problem domain that is more semantics-rich than the data categories modeled by topes. Thus, the level of semantics should be tuned to the problem domain. More is not always better.

From a software engineering standpoint, a vast gulf had remained largely unexplored between these semantics-rich modeling systems and the humble semantics-free grammars. This thesis research has not only identified a meaningful abstraction within this gulf, but it also has shown that capturing that abstraction in a simple model makes it possible to help people accomplish real-world goals. This result hints that software engineering researchers might be able to find other useful abstractions within the gulf of shallow semantics.

Toward future application-agnostic generalizations of research contributions

These four keys to topes' success offer signposts for how other researchers might be able to produce useful application-agnostic abstractions. Each key to topes' success provokes a question that researchers could ask of their own contributions: What aspects of this contribution are tied to the *real world*, and which therefore could potentially be captured in application-agnostic abstractions? To what extent could these abstractions be developed *incrementally*? How might *semi-automation* allow a user to benefit from such abstractions even if they are still incomplete and imprecise? Given the problem domain, what is the *semantic depth* required to capture useful information without resulting in unwieldy abstractions?

To illustrate the role of such questions as signposts for developing application-agnostic abstractions, consider the possibility of generalizing the numeric constraint models of Forms/3 beyond spreadsheets. These constraints not only take into account particular cell values, but also inter-cell dependencies resulting from formulas. One question is whether there is anything about a user's spreadsheet constraints *today* that would be worth capturing in an abstraction for reuse in checking spreadsheets *tomorrow*. Does a typical user make the same kinds of constraint-violation errors in spreadsheets day after day? Do groups of users (perhaps users who have similar kinds of work) make similar errors to one another? Do these users make the same kinds of errors even when they work in different applications (such as Excel, Access, or JavaScript)? If the answers to these questions are "yes", then it might be possible to develop a system that watches over a user or groups of users, learns what kinds of errors they commonly make in all the applications that they use, and then offers suggestions in all these applications based on inferred constraints.

The keys to topes' success provoke questions that could serve as signposts for moving such a research agenda forward. One signpost question could address whether there is anything in the *real world* that causes repeated errors in multiple applications.

For example, in healthcare industries, do novice users tend to make certain domain-specific dimension-unit errors when computing dosages in spreadsheets or other applications such as Medical Expression Language [82] or Microsoft Access)? In banking, do novice users tend to compute compound interest incorrectly? As a second signpost question, can such common errors be detected *incrementally*, to grow a domain-specific system that is increasingly smart about detecting errors? Or would it be necessary to set up a huge domain ontology before any common errors could be learned? As a third signpost question, how could *semi-automation* facilitate the training and invocation of a domain-specific system's capabilities, so that the system grows in accuracy and does not get out of control? As a final signpost question, given the nature of the problem domain, what is the appropriate *semantic depth* for capturing information about common errors? Modeling no semantics at all will probably fail to capture any information of interest, but modeling too much semantics could make the system hard to train, manage, and use.

The value of these signposts is that they highlight questions that might lead researchers to identify powerful new application-agnostic generalizations of their research contributions. Now, for a given research contribution, the answer might turn out to be "No, this particular contribution does not generalize across applications." But in other cases, the answer might be, "Yes, this contribution does generalize—and it might even transform the world some day."

References

[1] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets through Spatial Analyses. *2004 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004, 165-172.

[2] E. Allen, et al. The Fortress Language Specification, Version 1.0., Sun Microsystems, 2008.

[3] M. Anderberg. *Cluster Analysis for Applications*, Academic Press, 1973

[4] D. Appelt. Introduction to Information Extraction, *AI Communications (12)*, No. 3, 1999, 161-172.

[5] S. Bajracharya, T. Ngo, E. Linstead, and Y. Dou. Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search. *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, 2006, 6f81-682.

[6] V. Barnett and T. Lewis. *Outliers in Statistical Data,* Wiley, 1984.

[7] L. Barroso, J. Dean, and U. Hölze. Web Search for a Planet: The Google Cluster Architecture, *IEEE Micro (23)*, No. 2, 2003, 22-28.

[8] V. Basili, L. Briand, and W. Melo. A Validation of Object-Oriented Design Metrics as Quality-Indicators, *IEEE Transactions on Software Engineering (22)*, No. 10, 1996, 751-761.

[9] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web, Scientific American (284), No. 5, 2001, 34-43.

[10] H. Beyer and K. Holtzblatt. Contextual Design: Defining Customer-Centered Systems, Morgan Kaufmann, 1998.

[11] H. Bhasin. *Asp.NET Professional Projects*, Muska & Lipman Publishers, 2002.

[12] BEAs Bosworth: The World Needs Simpler Java, *eWeek Magazine*, February 19, 2004.

[13] T. Biggerstaff and C. Richter. Reusability Framework, Assessment, and Directions, *IEEE Software (4)*, No. 2, 1987, 41-49.

[14] S. Bikhchandani, D. Hirshleifer, and I. Welch. A Theory of Fads, Fashion, Custom, and Cultural Change as Informational Cascades, *Journal of Political Economy (100)*, No. 5, 1992, 992-1026

[15] A. Blackwell. First Steps in Programming: A Rationale for Attention Investment Models, *2002 IEEE Symposium on Human Centric Computing Languages and Environments*, 2002, 2-10.

[16] A. Blackwell. SWYN: A Visual Representation for Regular Expressions, *Your Wish Is My Command: Programming by Example*, 2001, 245-270.

[17] B. Boehm, et al. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0, *Annals of Software Engineering Special Volume on Software Process and Product Measurement*, J.C. Baltzer AG Science Publishers, 1995, 57-94.

[18] B. Boehm, et al. *Software Cost Estimation with COCOMO II*, Prentice-Hall, 2000.

[19] G. Boetticher, T. Menzies, T. Ostrand, and G. Ruhe. 4th International Workshop on Predictor Models in Software Engineering, *Companion to the 30th International Conference on Software Engineering*, 2008, 1061-1062.

[20] C. Bogart, M. Burnett, A. Cypher, and C. Scaffidi. End-User Programming in the Wild: A Field Study of CoScripter Scripts, *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2008, 39-46.

[21] J. Brandt, P. Guo, J. Lewenstein, and S. Klemmer. Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice, *4th International Workshop on End-User Software Engineering*, 2008, 1-5.

[22] G. Brat. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software, *Formal Methods in System Design (25),* No. 2, 2004, 167-200.

[23] Bureau of Labor Statistics. *Current Population Survey (CPS) Computer Ownership/Internet Usage Supplement*. Reports for all years may be retrieved from… www.census.gov/population/socdemo/computer/
while raw data for 1994 and later may be downloaded through the DataFerrett program available at…
dataferrett.census.gov/TheDataWeb/index.html

[24] J. Burkhardt and F. Détienne. An Empirical Study of Software Reuse by Experts in Object-Oriented Design. *5th International Conference on Human-Computer Interaction*, 1995, 38-138.

[25] M. Burnett, et al. End-User Software Engineering with Assertions in the Spreadsheet Paradigm. *Proc. 25th International Conference on Software Engineering*, 2003, 93-103.

[26] S. Chakrabarti. Mining the Web: Discovering Knowledge from Hypertext Data, Morgan Kaufmann, 2002.

[27] M. Coblenz, A. Ko, and B. Myers. Using Objects of Measurement to Detect Spreadsheet Errors. *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, 314-316.

[28] Cycorp. *Cyc Enhancement of Information Extraction*, whitepaper downloaded April 30, 2009, http://www.cyc.com/cyc/technology/whitepapers_dir/IE-Improvement-Whitepaper.pdf

[29] P. Drucker. *The Age of Discontinuity*, Transaction Publishers, 1992.

[30] E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003.

[31] M. Erwig and M. Burnett. Adding Apples and Oranges. 4th International Symposium on Practical Aspects of Declarative Languages, 2002, 173-191.

[32] N. Fenton and M. Neil. A Critique of Software Defect Prediction Models. IEEE Transactions on Software Engineering, (25), No. 5, 1999, 675-689.

[33] M. Fisher II and G. Rothermel. The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms, Technical Report 04-12-03, University of Nebraska—Lincoln, 2004.

[34] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures,* PhD Thesis, Department of Information and Computer Science, University of California - Irvine, 2000.

[35] J. Fujima, A. Lunzer, K. Hornboek, and Y. Tanaka. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access,

17th Annual ACM Symposium on User Interface Software and Technology, 2004, 175-184.

[36] E. Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[37] T. Green and M. Petre. Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *Journal of Visual Languages and Computation (7)*, No. 2, 1996, 131-174.

[38] N. Gulley. Improving the Quality of Contributed Software and the MATLAB File Exchange, *2nd Workshop on End User Software Engineering*, 2006, 8-9.

[39] J. Hall. A Risk and Control-Oriented Study of the Practices of Spreadsheet Application Developers. *29th Hawaii International Conference on System Sciences*, 1996, 364-373.

[40] D. Halbert. SmallStar: Programming By Demonstration in the Desktop Metaphor, *Watch What I Do: Programming By Demonstration*, 1993, 102-123.

[41] D. Hecker. Occupational Employment Projections to 2012, *Monthly Labor Review* (*127)*, No. 2, 2004, 80-105.

[42] S. Hipple and K. Kosanovich. Computer and Internet Use at Work in 2001, *Monthly Labor Review* (*126)*, No. 2, 2003, 26-35.

[43] R. Hoffmann, et al. Amplifying Community Content Creation with Mixed Initiative Information Extraction, *27th International Conference on Human Factors in Computing Systems*, 2009, 1849-1858.

[44] D. Huynh, R. Miller, and D. Karger. Potluck: Data Mash-Up Tool for Casual Users. *Lecture Notes in Computer Science (4825)*, 2007, 239-252.

[45] B. Keller and R. Lutz. Evolving Stochastic Context-Free Grammars from Examples Using a Minimum Description Length Principle. *1997 Workshop on Automata Induction Grammatical Inference and Language Acquisition*, 1997, http://citeseer.ist.psu.edu/59698.html

[46] A. Kennedy. *Programming Languages and Dimensions*, PhD thesis, St. Catharine's College, University of Cambridge, 1996.

[47] R. Khare and T. Çelik. Microformats: A Pragmatic Path to the Semantic Web. *15th International World Wide Web Conference*, 2006, 865-866.

[48] J. Kim and C. Mueller. Factor Analysis: Statistical Methods and Practical Issues, Sage Publications, 1978.

[49] D. Knuth. Semantics of Context-Free Languages, *Journal of Mathematical Systems Theory (2)*, No. 2, 1968, 127-146.

[50] A. Ko, B. Myers, and H. Aung. Six Learning Barriers in End-User Programming Systems, *2004 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004, 199-206.

[51] A. Ko, et al. The State of the Art in End-User Software Engineering, Submitted to *ACM Computing Surveys*, 2008.

[52] A. Koesnandar. Using Assertions to Help End-User Programmers Create Dependable Web Macros, *16th International Symposium on Foundations of Software Engineering*, 2008, 124-134.

[53] M. Lanza, M. Godfrey, and S. Kim. 5th Working Conference on Mining Software Repositories, *Companion to 30th International Conference on Software Engineering*, 2008, 1037-1038.

[54] J. Lawrance, R. Abraham, M. Burnett, and M. Erwig. Sharing Reasoning about Faults in Spreadsheets: An Empirical Study, *2006 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006, pp. 35-42.

[55] D. Lenat. CYC: A Large-Scale Investment in Knowledge Infrastructure, *Communications of ACM*, *(38)*, No. 11, 1995, 33-38.

[56] D. Lenat. From 2001 to 2001: Common Sense and the Mind of HAL, *HAL's Legacy: 2001's Computer as Dream and Reality*, MIT Press, 2001, 193-209.

[57] D. Lenat, R. Guha, K. Pittman, and D. Pratt. Cyc: Toward Programs with Common Sense, *Communications of ACM, (33)*, No. 8, 1990, 30-49.

[58] A. Lenhart and S. Fox. *Twitter and Status Updating*, Pew Internet & American Life Project, 2009.

[59] K. Lerman, S. Minton, and C. Knoblock. Wrapper Maintenance: A Machine Learning Approach, *Journal of Artificial Intelligence Research (18)*, 149-181.

[60] G. Leshed, E. Haber, T. Matthews, and T. Lau. CoScripter: Automating & Sharing How-To Knowledge in the Enterprise, *26th SIGCHI Conference on Human Factors in Computing Systems*, 2008, 1719-1728.

[61] H. Lieberman, H. Liu, P. Singh, and B. Barry. Beating Common Sense into Interactive Applications, *AI Magazine*, *(25)*, No. 4, 2004, 63-76.

[62] H. Lieberman, B. Nardi, and D. Wright. Training Agents to Recognize Text by Example, *3rd Annual Conference on Autonomous Agents*, 1999, 116-122.

[63] J. Lin, et al. End-User Programming of Mashups with Vegemite, *2009 International Conference on Intelligent User Interfaces*, 2009, 97-106.

[64] G. Little, et al. Koala: Capture, Share, Automate, and Personalize Business Processes on the Web, *2007 SIGCHI Conference on Human Factors in Computing Systems*, 2007, 943-946.

[65] H. Liu and P. Singh. *OMCSNet: A Commonsense Inference Toolkit*, whitepaper from MIT, downloaded April 30, 2009, http://web.media.mit.edu/~hugo/omcsnet/omcsnet.pdf

[66] M. Madden and S. Jones. *Networked Workers*, Pew Internet & American Life Project, 2008.

[67] E. Marsh and D. Perzanowski. MUC-7 Evaluation of IE Technology: Overview of Results, *7th Message Understanding Conference*, 2001.

[68] R. McDaniel and B. Myers. Getting More Out of Programming-By-Demonstration, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1999, 442-449.

[69] T. Menzies, et al. Problems with Precision: A Response to Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors', *IEEE Transactions on Software Engineering (33)*, No. 9, 2007, 637-640.

[70] R. Miller. *Lightweight Structure in Text*, PhD thesis, Computer Science Department, Carnegie Mellon University, 2002.

[71] R. Miller and B. Myers. *Creating Dynamic World Wide Web Pages by Demonstration*, Technical Report CMU-CS- 97-131, School of Computer Science, Carnegie Mellon University, 1997.

[72] T. Mitchell. *Machine Learning*, McGraw-Hill, 1997.

[73] F. Modugno and B. Myers. Pursuit: Graphically Representing Programs in a Demonstrational Visual Shell, *Proceedings of the CHI '94 Conference Companion on Human Factors in Computing Systems*, 1994, 455-456.
[74] P. Mohagheghi, R. Conradi, O. Killi, and H. Schwarz. An Empirical Study of Software Reuse vs. Defect-Density and Stability, *26th International Conference on Software Engineering*, 2004, 282-291.
[75] R. Moser, W. Pedrycz, and G. Succi. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction, *30th International Conference on Software Engineering*, 2008, 181-190.
[76] F. Mosteller and C. Youtz. Quantifying Probabilistic Expressions, *Statistical Science (5)*, No. 1, 1990, 2-12.
[77] B. Myers, J. Pane, and A. Ko. Natural Programming Languages and Environments, *Communications of ACM (47)*, No. 9, 2004, 47-52.
[78] B. Nardi. A Small Matter of Programming: Perspectives on End User Computing, MIT Press, 1993.
[79] B. Nardi, J. Miller, and D. Wright. Collaborative, Programmable Intelligent Agents, *Communications of the ACM (41)*, No. 3, 1998, 96-104. See http://www.miramontes.com/writing/add-cacm/index.html for screenshots.
[80] R. Nix. Editing By Example, ACM Transactions on Programming Language Systems (7), No. 4, 1985, 600-621.
[81] D. Norman. *Psychology of Everyday Things*, Basic Books, 1988.
[82] E. Orrick. Position Paper for the CHI 2006 Workshop on End-User Software Engineering, *2nd Workshop on End User Software Engineering*, 2006, 10-11.
[83] X. Ou, et. al. *Dynamic Typing with Dependent Types*, Technical Report TR-695-04, Department of Computer Science, Princeton University, 2004.
[84] J. Pane. Designing a Programming System for Children with a Focus on Usability, *Conference on Human Factors in Computing Systems*, 1998, 62-63.
[85] R. Panko. What We Know About Spreadsheet Errors, *Journal of End User Computing (10)*, No. 2, 1998, 15-21.
[86] M. Petre and A. Blackwell. Children as Unwitting End-User Programmers, *2007 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2007, 239-242.
[87] B. Pierce. Types and Programming Languages, MIT Press, 2002.
[88] M. Porter. An Algorithm for Suffix Stripping, *Program (14)*, No. 3, 1980, 130-137.
[89] A. Pras and J. Schoenwaelder. RFC 3444: On the Difference between Information Models and Data Models, IETF, 2003.
[90] E. Rahm and H. Do. Data Cleaning: Problems and Current Approaches, *IEEE Data Engineering Bulletin (23),* 4, 2000, 3-13.
[91] V. Raman and J. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System, *27th International Conference on Very Large Data Bases*, 2001, 381-390.
[92] *Random House Unabridged Dictionary*, Random House, 2006, http://www.dictionary.com
[93] O. Raz, P. Koopman, and M. Shaw. Semantic Anomaly Detection in Online Data Sources, *24th International Conference on Software Engineering*, 2002, 302-312.

[94] M. Resnick, M. Flanagan, C. Kelleher, and M. MacLaurin. Growing Up Programming: Democratizing the Creation of Dynamic, Interactive Media, *27th International Conference Extended Abstracts on Human Factors in Computing Systems*, 2009, 3293-3296.

[95] J. Rode. *Web Application Development by Nonprogrammers: User-Centered Design of an End-User Web Development Tool*. PhD thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, 2005.

[96] J. Rode, E. Toye, and A. Blackwell. The Fuzzy Felt Ethnography—Understanding the Programming Patterns of Domestic Appliances, *Journal of Personal and Ubiquitous Computing (8)*, No. 4, 2004, 161-176.

[97] E. Rogers. *Diffusion of Innovation*, Simon & Schuster, 1995.

[98] M. Rosson, J. Ballin, and H. Nash. Everyday Programming: Challenges and Opportunities for Informal Web Development, *2004 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004, 123-130.

[99] M. Rosson and J. Carroll. The Reuse of Uses in Smalltalk Programming, *Transactions on Computer-Human Interaction (3)*, No. 3, 1996, 219-253.

[100] C. Scaffidi. Unsupervised Inference of Data Formats in Human-Readable Notation, *9th International Conference on Enterprise Information Systems - HCI Volume*, 2007, 236-241.

[101] C. Scaffidi, C. Bogart, M. Burnett, A. Cypher, B. Myers, and M. Shaw. Predicting Reuse of End-User Web Macro Scripts. Submitted to *2009 IEEE Symposium on Visual Languages and Human Centric Computing*.

[102] C. Scaffidi, A. Cypher, S. Elbaum, A. Koesnandar, J. Lin, B. Myers, and M. Shaw. Using Topes to Validate and Reformat Data in End-User Programming Tools, *4th Workshop on End-User Software Engineering*, 2008, 11-15.

[103] C. Scaffidi, A. Cypher, S. Elbaum, A. Koesnandar, and B. Myers. Scenario-Based Requirements for Web Macro Tools, *2007 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2007, 197-204.

[104] C. Scaffidi, A. Ko, B. Myers, M. Shaw. Dimensions Characterizing Programming Feature Usage by Information Workers, *2006 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006, 59-62.

[105] C. Scaffidi, B. Myers, and M. Shaw. Fast, Accurate Creation of Data Validation Formats by End-User Developers, *2nd International Symposium on End-User Development*, 2009, to appear.

[106] C. Scaffidi, B. Myers, and M. Shaw. Intelligently Creating and Recommending Reusable Reformatting Rules, *14th International Conference on Intelligent User Interfaces*, 2009, 297-306.

[107] C. Scaffidi, B. Myers, and M. Shaw. Tool Support for Data Validation by End-User Programmers, *Companion to 30th International Conference on Software Engineering*, 2008, 867-870.

[108] C. Scaffidi, B. Myers, and M. Shaw. Topes: Reusable Abstractions for Validating Data, *30th International Conference on Software Engineering*, 2008, 1-10.

[109] C. Scaffidi, B. Myers, and M. Shaw. Trial By Water: Creating Hurricane Katrina "Person Locator" Web Sites, *Leadership at a Distance: Research in Technologically-Supported Work*, Lawrence Erlbaum Publishers, 2007, 209-222.

[110] C. Scaffidi, M. Shaw, and B. Myers. Estimating the Numbers of End Users and End User Programmers, *2005 IEEE Symposium on Visual Languages and Human-Centric*, 2005, 207-214.

[111] J. Schlimmer and L. Hermens. Software Agents: Completing Patterns and Constructing User Interfaces, *Journal of Artificial Intelligence Research (1)*, 1993, 61-89.

[112] J. Segal. *Professional End User Developers and Software Development Knowledge,* Technical Report 2004/25, Department of Computing, Faculty of Mathematics and Computing, The Open University, Milton Keynes, United Kingdom, 2004.

[113] M. Shaw. An Input-Output Model of Interactive Systems, *1986 SIGCHI Conference on Human Factors in Computing Systems*, 1986, 261-273.

[114] M. Shaw. Larger Scale Systems Require Higher-Level Abstractions, *5th International Workshop on Software Specifications and Design*, 1989, 143-146.

[115] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, *ACM Computing Surveys (22*), No. 3, 1990, 183-236.

[116] F. Shull, et al. What We Have Learned About Fighting Defects, *8th IEEE International Software Metrics Symposium*, 2002, 249-258.

[117] G. Stahl, T. Sumner, and A. Repenning. Internet Repositories for Collaborative Learning: Supporting Both Students and Teachers, *1st International Conference on Computer Support for Collaborative Learning*, 1995, 321-328.

[118] J. Stylos, B. Myers, and A. Faulring. Citrine: Providing Intelligent Copy-And-Paste, *17th Annual ACM Symposium on User Interface Software and Technology*, 2004, 185-188.

[119] J. Teng, V. Grover, and W. Güttler. Information Technology Innovations: General Diffusion Patterns and Its Relationships to Innovation Characteristics, *Transactions on Engineering Management (49)*, No. 1, 2002, 13-27.

[120] M. Tomita. An Efficient Augmented-Context-Free Parsing Algorithm, *Journal of Computational Linguistics (13),* No. 1-2, 1987, 31-46.

[121] R. Walpole and M. Burnett. Supporting Reuse of Evolving Visual Code, *1997 IEEE Symposium on Visual Languages*, 1997, 68-75.

[122] A. Westerinen, et al. RFC 3198: Terminology for Policy-Based Management, IETF, 2001.

[123] S. Wiedenbeck. Facilitators and Inhibitors of End-User Development by Teachers in a School Environment. *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, 215-222.

[124] N. Zang, M. Beth Rosson, and V. Nasser. Mashups: Who? What? Why?. 2008 SIGCHI Conference on Human Factors in Computing Systems - Work-in-Progress Posters, 2008, 3171-3176.

[125] L. Zadeh. *Fuzzy Logic*, Technical Report CSLI-88-116, Stanford University, 1988.

[126] H. Züllighoven. *Object-Oriented Construction Handbook*, Morgan Kaufmann, 2003.