

No application is an island: Using topes to transform strings during data transfer

Atipol Asavametha, Prashanth Ayyavu, Christopher Scaffidi

School of Electrical Engineering and Computer Science

Oregon State University

Corvallis, OR, USA

{asavamat, ayyavu, cscaffid} @ eecs.oregonstate.edu

Abstract—Users and programmers frequently need to move information between applications, including desktop and web applications. Transferring data often involves reformatting strings such as phone numbers or extracting parts from them, but actually performing these transformations typically requires tedious, error-prone operations. For example, programmers must write messy code to parse and reformat strings passed between web services, while ordinary end users must manually reformat strings that they want to copy-paste between applications.

In this paper, we show at an architectural level how topes can be used to smooth the flow of data between applications by automatically transforming strings on demand. We have demonstrated the generality and usefulness of this approach by using topes to automatically transform data moving between applications, web sites and web services, thereby showing how topes can make it simpler for both end users and programmers to transfer information between applications.

Keywords- Content convergence; data transformation

I. INTRODUCTION

People transfer data between applications remarkably often. Studies of end users have revealed many tasks that involved transferring data [10]. For example, computing per diem rates for travel reimbursements often requires copying dates and place names from spreadsheets to web sites, while creating staff rosters commonly involves copying names, phone numbers and other data from web sites to documents. Other common tasks involve transferring data from web sites to web sites, applications to applications and documents to documents.

Data transfer often requires intervening transformation of values, since different applications may represent data in different ways—a problem referred to as “data heterogeneity.” Transformation commonly involves reformatting strings from one format to another or extracting pieces of strings [10]. For example, an end user might need to reformat “JOHN DOE” to “Doe, John” or might extract the given and family names and paste them into separate fields on a web form. Browsers and applications currently cannot automate these reformatting and extraction operations, so end users must instead perform them through tedious, error-prone edits.

Programmers face similar challenges when writing programs to transfer data from one application to another. For example, after Hurricane Katrina, several teams set up web

sites where survivors could report their locations [10]. Other teams tried to create programs to read the data off those web sites, to parse values, to reformat them into consistent formats, and to store them in a consolidated database. Unfortunately, no library or API was available for automating this parsing and reformatting, so programmers had to write this code by hand, which proved to be complex and was only partially completed.

Researchers and companies have tackled problems like these in a piecemeal fashion. Some have focused on data integration in a *database setting*, for example by inventing an algorithm that reformats strings on the fly so tables can be joined [5]. Others have focused on the *user interface setting*, for example by developing an enhanced clipboard that parses strings, extracts parts, and pastes parts into different textboxes [13].

We see data heterogeneity in these settings as instances of a more general architectural problem. Specifically, there should be some generalized way to automate reformatting and extraction when transferring individual strings from one application to another, regardless of whether those applications are web sites, databases, spreadsheets, custom desktop applications, or *any* other kind of application. Complex setting-specific operations (such as table joins) could then be built on top of this generalized approach for transforming individual data values.

In the current paper, we show how this general architectural problem can be solved using the topes data model and its supporting toolset [11]. A tope is a package of functions for recognizing and transforming values in one data category. For example, a user might create a tope for person names, then use it to reformat strings as they are transferred from a spreadsheet or a web site to a desktop application. In prior work, topes have proven useful for transforming data *within* individual applications, but we had not previously considered transformation of data during transfer *between* applications.

This paper’s contribution is thus to describe where, when and how topes can be used to automate string reformatting and extraction operations during data transfer. We characterize our approach’s applicability by qualitatively analyzing how topes can be applied in five common architectural styles. We test our approach’s practical usefulness through the case study evaluation method [15], using three cases that explore how topes can simplify data transfer for users and programmers.

These studies revealed that topes were indeed able to perform the string transformations needed in a variety of situations, without causing significant difficulties along the way. We found that using topes simplified the code required to perform string transformations. In the specific setting of retrieving data from web services and web applications, we found that topes actually made it possible to *completely insulate* client applications from changes in data formats, at least at the level of individual string values, thereby providing full *forward and backward compatibility* in the formatting of individual string data values.

This paper is organized as follows. Section II reviews related work, which has addressed data heterogeneity in particular limited settings rather than in a more abstract, generalizable manner. Section III summarizes the topes model and related preliminary work. Section IV presents the core architectural problem involved in data heterogeneity, and it describes our approach for using topes in a range of architectural styles. Section V presents our case study evaluation methodology, while Section VI describes our evaluation's results showing the practical usefulness of topes. Section VII summarizes key conclusions and future work.

II. RELATED WORK

When applications contain similar data but represent it differently, then transferring data between them requires transforming data between representations. At the level of individual values, this problem is called "*data heterogeneity*" [6][8], which frequently involves character strings. For example, "there are often multiple ways of referring to companies (e.g., IBM vs. International Business Machines), people names (that are often incomplete), and addresses" [6]. If one application internally uses "IBM" to reference that company, but another application internally uses "International Business Machines", then one string must be transformed to the other when transferring data between applications.

In addition to differences in the representation of *individual* values, two applications can differ in how they combine values into hierarchical *structures*. This problem, called "*schema heterogeneity*" [6][8], has attracted decades of research in many settings including relational databases, XML, web services, and the semantic web (e.g., [1][3][4][5][16]). Some of these approaches for overcoming schema heterogeneity are extremely sophisticated. For example, the SMA matching algorithm helps to overcome schema heterogeneity between web services by using the WordNet knowledge base in order to efficiently align web service parameter structures based on semantics [16]. In general, these approaches for overcoming schema heterogeneity are limited to one particular setting (databases/web services/semantic web). The reason is that each setting's data has a certain hierarchical shape; consequently, associated structural transformations are specific to particular settings. For example, transferring data between databases with different representations involves transforming tabular tuple structures, while transferring XML data involves reshaping and merging tree-shaped structures.

Most approaches for automating structural transformations also include some modicum of support for transforming individual strings, and vice versa. For example, string transformation is needed to join database tables that represent keys differently [5]. Conversely, some approaches automate transformation of individual strings in particular settings and provide incidental support for structural transformations. For example, Citrine allows users to copy a string and paste pieces of it into multiple textboxes [13]. Citrine can be viewed as overcoming data heterogeneity, in that each output is computed by parsing the input string and extracting part of it; Citrine also achieves some small structural transformation, in the sense that one string is used to compute several.

Where all these approaches come up short is that they are setting-specific (or limited to integration of two settings, such as importing XML into databases). Because of the setting-specificity of structural transformations, it is doubtful that any single model of structural transformations could generalize over all settings. *However, generalizability could and should be pursued at the level of individual values.* This is the problem addressed in this paper.

For example, there is no way to express that "IBM" is synonymous with "International Business Machines", and then to apply this rule in many settings, such as for transferring between databases, between spreadsheets, or between textboxes. A programmer could create a rule in SQL, for instance, to reformat company names as they are read from one database to another, but the platform-specificity of SQL would prevent using this rule to transform company names as they are read from a web site into Microsoft Word. As another example, it should be possible to create and execute rules for reformatting and extracting pieces of a person name regardless of what applications are on the sending or receiving end of a data transfer. In other words, transformation rules are currently "locked up," so to speak, in setting-specific rules and notations, with no reusability across settings. But the rules for individual strings' reformatting and extraction operations depend on the information domain, not from the fact that the strings are in a database or a textbox. Therefore, these rules *should* be reusable across different settings.

We believe that our topes model can provide an approach for addressing this limitation, by providing an application-independent mechanism for transforming strings as they are transferred between applications [11]. Our prior work examined whether topes could be used to transform data *within* a particular application. In the current paper, we assess whether topes can be used to solve this new problem of transforming strings as they are transferred *between* applications. Using topes for this new purpose introduces a coordination problem not present in a single application: when will a tope be invoked during a data transfer, and by which application(s), and how will end users or programmers control the process? Using topes for this new problem requires not only answering these questions, but giving answers that generalize over a broad range of different settings.

III. PRELIMINARY WORK: TOPES AND THE TDE

A tope is an abstraction that describes strings in a data category independently of any particular setting [11]. For example, one tope might describe phone numbers, while another might describe person names.

A tope is a directed graph, where each node corresponds to a format and each edge corresponds to a reformatting rule between formats. For each node, a tope contains a function to parse strings that match that format; for each edge, a tope contains a function to reformat a string from one format to another. For example, a tope for person names might hypothetically have four formats matching strings like “John von Neumann”, “JOHN von Neumann”, “von Neumann, JOHN”, and “von Neumann, John”. The parsing functions would look for the presence of spaces and commas to determine where best to split a string, and they would include rules about how to handle ambiguity (e.g., is “von” part of the first or last name?) The reformatting functions would reorder parts, change separators and modify capitalization.

A. Using the TDE to create topes

Implementing and testing such a tope by hand in a language like JavaScript would be very time-consuming (especially since person names can be written in more than four formats). Therefore, the past few years have been devoted to developing a toolset called the Tope Development Environment (TDE) that enables end users (or programmers) to quickly and correctly create topes for an enormous range of different kinds of strings, as well as to invoke topes within specific settings such as spreadsheets [11]. To create a tope, a user can provide examples of strings (perhaps in a variety of formats), from which the TDE infers a boilerplate tope. The user can then test

and customize this tope as needed until the rules are tuned satisfactorily (Figure 1). These rules can be specified as always, often, rarely, or never true; thus, it is possible for a tope to identify questionable strings that deserve to be double-checked but which still might be valid. (We recently augmented the TDE so it supports custom functions written in JavaScript, thereby covering full Turing completeness in parsing and reformatting strings. In practice, however, the TDE’s form-based editor in Figure 1 suffices for all but the most complex topes [11].)

We have provided “plug-ins” to augment several applications so users can invoke a tope’s operations in each application. For example, if a user assigns a tope to a column in a spreadsheet, Excel passes these strings to topes in the TDE to parse them; for strings that could not be parsed completely, Excel flags those cells for user review. The user can also direct Excel to use a tope to reformat all of the strings in a spreadsheet column. The TDE programmatically exposes parts of parsed strings (e.g., it is possible to retrieve the day, month and year of a date), though our plugin currently does not make use of these. An experiment has shown that topes enable users to reformat strings in a spreadsheet so quickly that the cost of creating a tope is “paid off” after reformatting 47 strings [11].

B. Limitations to prior research

While our prior research established that topes are useful for manipulating data *within* an application, we have not yet carefully evaluated the usefulness of topes for manipulating data flowing *between* applications. The cross-application context presents a different problem than the one we have previously explored because this new setting introduces additional complexities. For example, when a programmer

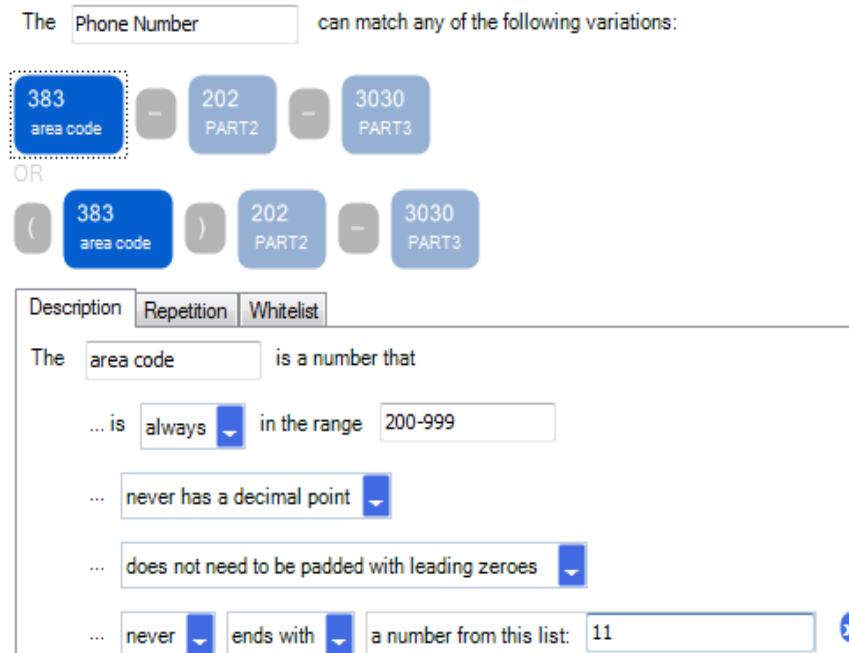


Figure 1. Editing constraints on the “area code” part of a phone number tope. The other parts have not yet been named, nor have their constraints been tuned. From this description of formats, the TDE generates a tope’s parsing and reformatting rules.

creates an application Y that reads data from another application X, the programmer might not have access to the source code of application X, which will prevent the programmer from modifying X so that it returns data in the format needed by Y. In some cases, as the next section discusses in detail, the programmer might not have access to X or Y—how then can the programmer invoke a tope to transform data during transfer from X to Y? In short, while prior work examined whether topes improve ease of data manipulation within applications, this prior work did not explore how topes might be used to transform strings during data transfer—the problem to which we now turn.

IV. USING TOPES TO TRANSFORM STRINGS DURING DATA TRANSFER

Software architecture provides a framework for exploring the way in which topes can be used to transform strings during data transfer. In software architecture, applications and systems of applications are described in terms of elements called components and connectors [14]. Components are self-contained elements that ultimately implement an application’s core functionality. They are usually accessible through a programmatic interface, though they may also provide a visual interface for user interaction. Components transfer data to one another through connectors. Examples of connectors include method calls, the operating system clipboard, and network messages.

Any time that one component passes data to another, data heterogeneity can become a problem. In general, a component X might internally represent a data value v with format F_X , but v might need to be transferred via connector C to a component Y that internally uses format F_Y to represent v (Figure 2). Thus, v must be transformed from F_X to F_Y so Y can use it.

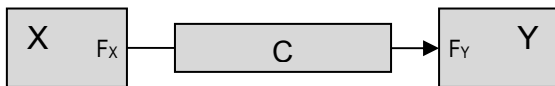


Figure 2. Architecture exhibiting data heterogeneity

A pair of heterogeneous components and a connector will often be embedded in a larger architecture, within which other components might also exhibit other pair-wise heterogeneity. For example, many user tasks observed in empirical studies required reading data from a spreadsheet X, pasting the data into a web site Y (and submitting a web form), then copying data from Y and pasting this data into another site Z [10]. When performing these tasks, users typically issue copy-paste keystrokes causing the operating system clipboard (and a browser) to act as a connector C that transfers data from X to Y and Y to Z. Unfortunately, X and Y often use different formats to represent each value, so the first data transfer frequently requires the user to manually perform reformatting or extraction operations after pasting into the browser (before submitting the form to Y). Copying from Y to Z requires similar edits if Y and Z use different formats.

Our goal is to show where, when and how topes can be used to automate these operations, thereby addressing data

heterogeneity and simplifying data transfer by eliminating the need for this tedious manual work.

A. Loci for transforming with topes

Casting data heterogeneity as an architectural problem, as above, reveals the three primary places where heterogeneity can be corrected: at the data sender X, within the connector C, and at the data receiver Y. Each could serve as a locus for invoking a tope in order to automate string transformation.

1) Locus #1-Transform data with topes before transfer

While X might internally format v as F_X , it is often possible to modify X so it invokes a tope to transform v to F_Y just before transfer to Y. For example, an application X might internally format a phone number like “(888) 555-1212” but invoke a tope to reformat it to “1-888-555-1212” just before sending it to a web service Y for checking voice mail. If the programmer of X delegates the responsibility of reformatting v to a reusable tope, there would be no need to manually re-implement that transformation’s code in every setting.

2) Locus #2-Transform data with topes during transfer

One advantage of calling out connectors as first-class architectural elements is that doing so emphasizes that connectors can be complex and active rather than simple and passive—they can execute operations such as buffering and transformation [14]. In particular, C could accept strings in F_X and produce strings in F_Y by invoking a tope along the way. That way, for example, an application X could transmit “(888) 555-1212”, and a receiving web service Y could receive data in the preferred format “1-888-555-1212”.

3) Locus #3-Transform data with topes after transfer

Finally, if Y receives data in an undesirable format, then it could pass the strings to a tope for transformation. This would allow Y to receive strings in *any* format recognized by the tope. For example, Y could accept strings like “888-555-1212”, “(888) 555-1212”, or any other format known to the tope because, upon receipt of a string, it could be reformatted on demand into a string like “1-888-555-1212” or whatever format Y requires. Y would no longer be bound to a particular syntactic input, but rather to a semantic input.

B. Compatibility with standard software architectures

While each of these loci theoretically might serve as a place for invoking topes, in practice we believe that it is harder to use topes in some situations than in others. To explore this issue, we qualitatively analyze how topes might be used in five standard architectural styles. Architectures with a similar style use components and connectors in similar ways [14]. Thus, each style presents particular challenges to using topes. In particular, the applicability of our approach in a specific architecture hinges on whether programmers and end users typically have control over each component and connector. If it is not possible for a user or programmer to control a particular element in a style, then it is not possible to make that element invoke a tope.

1) Client-server architectural style

In a client-server style, a server component waits for and responds to requests from clients, usually via a standardized protocol [14]. Most existing approaches for structural or data transformation (Section II) involve a client-server architecture. In the web service setting, for example, clients call web services via SOAP or REST connector [16]. In a database setting, an RDBMS awaits SQL requests [3].

When programmers implement either clients or servers, they typically have complete control over that code, making loci #1 and #3 natural choices for using topes to overcome data heterogeneity. (The data receiver could be the client or the server, of course.) However, because the connector is usually standardized, its code is often not visible or easily modifiable by ordinary programmers, making locus #2 impractical. For example, reviewing the documentation for Microsoft’s SOAP implementation suggests that customizing it to invoke a tope would require writing dozens or hundreds of lines of code.

The client often has a user interface, so it could ask users how to transform strings before sending data to the server (locus #1), and the client could record those instructions in order to avoid bothering the user when using that server again. Servers and connectors in a client-server style rarely have a user interface, limiting the applicability of #2 and #3 for users.

2) Repository architectural style

In the repository style, a server-like component called a repository is responsible for storing data. Other components, like clients, read from the repository, perform operations, and perhaps write results back. (In one specialization of this style, called the blackboard style, the clients work together on a common problem by solving sub-problems [14].) While client-server is the dominant style in most settings addressed by existing approaches (Section II), a repository sometimes appears. For example, a web service can serve as a repository between clients: one client can send data to another by uploading data to a web service, from which the other client downloads the data. Thus, the repository is essentially a sophisticated connector between clients (as well as a server).

When programmers create a component X that provides data to another Y via a repository connector R, they may have access to the code of X, Y and R. In this case, they can use topes at all three loci. Both X and Y typically have a user interface, making loci #1 and #3 feasible for users. However, R rarely has a user interface, except when the operating system’s clipboard acts as a repository between applications. In this one case, the operating system could be augmented with a window enabling users to invoke topes during data transfer (locus #2).

3) Interpreter architectural style

The interpreter style has three main components: a textual script S, an interpreter I that parses the script, and a component T that I manipulates as it steps through S [14]. For example, S could be a spreadsheet, I could be Excel, and T could be a 2-dimensional array of numbers; I parses formulas in the cells of S, reads values from S or T, computes, and stores results in T.

The three components are typically connected through calling of procedures or reading/writing of memory.

Focusing on the S-I data transfer, S is a file rather than an executable, so it cannot directly invoke a tope, limiting the usefulness of locus #1. The procedure call or memory read/write in the connector is too primitive to be modified, limiting locus #2. Locus #3 could be used by programmers or by end users. For example, when I receives strings like “888-555-1212” and “(888) 555-1212”, it could use a tope to transform these into a canonical format preferred by I’s programmer. This format could also be made configurable, perhaps by allowing an end user to textually specify in S what canonical format I should use.

Shifting focus to the I-T data transfer, locus #1 could be used in a similar manner to transform data before transfer to T. Again, the connector’s primitiveness limits the usefulness of #2. When procedures of T are called to store results, #3 might be useful to programmers, but T sometimes is just a data structure lacking executable code, and it normally lacks a user interface, limiting the usefulness of #3 in these cases.

4) Object-oriented architectural style

In the object-oriented style, components are instances of classes that call one another via methods [14]. These components are under the control of programmers and often have user interfaces, making #1 and #3 feasible for both programmers and end users. Method invocation is low-level and lacking in a user interface, limiting the usefulness of #2.

5) Peer-to-peer architectural style

Peer-to-peer is very similar to client-server, except that either component can initiate the connection, and both usually have a user interface [14]. As such, loci #1 and #3 are suitable for both programmers and end users, with #2 still being much less useful.

Table 1 summarizes this analysis. Loci #1 and #3 have much wider applicability than #2, ultimately because programmers and users generally have more control over components than over connectors.

Table 1. Applicability of topes under Programmer and User control (uppercase=strong applicability, lowercase=weak) for 15 combinations of locus and architectural style.

Architectural Style	Locus		
	#1 before transfer	#2 during transfer	#3 after transfer
Client-server	PU		P
Repository	PU	Pu	PU
Interpreter (S-I / I-T)	/ PU	/	PU / p
Object-oriented	PU		PU
Peer-to-peer	PU		PU

V. EVALUATION METHODOLOGY

To test the practical usefulness of our approach for transforming strings during data transfer, we applied the widely-practiced case study evaluation method [15]. The case study method is particularly appropriate for exploring how a

technology or technique integrates or interacts with complexities faced in real world problems. This strongly contrasts with controlled experiments, such as laboratory studies, which can make fine measurements but which necessarily must be simplified versions of real world complexity. Therefore, we used a qualitative case study method because we wanted to understand the practical problems that might be encountered with using our approach in practice. For case studies to be valid tests of a technology or idea, they should be driven by propositions, which are qualitative analogues of the statistical hypotheses used in experiments [15]. Our case studies tested the following propositions:

- 1) *Most of the difficulties encountered will result from technologies other than topes or the TDE.*
- 2) *Topes will be able to perform the string transformations needed in a variety of situations.*
- 3) *Topes will be useful at all three loci (before/during/after data transfer), though not necessarily in every combination of locus and architectural style.*
- 4) *Using topes will simplify the code required to perform string transformations.*

To test these four propositions, we conducted three case studies (A-C), each of which aimed to integrate topes into a particular application or class of applications. In particular, we attempted to use topes to transform strings during data transfer (A) from application to application through the operating system clipboard, (B) from website to website through a scripting environment, and (C) from web services or sites to client applications through http. Of these case studies, part of one was summarized in a prior workshop paper [12], while the other two are completely new. We selected these study settings because they demonstrate a variety of architectural styles, and they cover a broad set of different settings ranging from user-facing applications (A) all the way to client-server applications with potentially complex, networked interactions (C). Using notes, screenshots, email, and weekly meetings, we tracked our successes and problems along the way. At the conclusion of the study, we reflected on the process, in order to characterize the strengths and weaknesses of our approach.

Because our propositions focused on topes' capabilities, rather than usability, we did not recruit human subjects to perform the studies, but rather we performed the studies ourselves. Case study A was done by the PI (Scaffidi), who had previously invented topes and was most familiar with the TDE. Case study B was done together by our team's masters students (Asavametha and Ayyavu), who had no prior experience with topes previous to this work, but who spent approximately 1 month prior to the studies becoming familiar with topes and the TDE. Case study C was done by all three of us, each focusing on a different aspect of the case study. We each had prior experience as professional software engineering consultants or researchers. Each of us had approximately 4-7 years of experience with Java, C#, and the other programming languages required for these studies, as well as comparable experience with web services, http, XML, and the other technologies required for data transfer and representation.

In short, we believe that our studies are a valid test of topes' usefulness in the hands of programmers who have a moderate amount of experience with mainstream programming technologies, and who have taken the time to become experienced with topes and the TDE.

VI. RESULTS

We present the results of our three case studies below. As discussed further by the next section, we found strong support for all four propositions, indicating that topes do provide a useful way to automate string reformatting and extraction operations during data transfer.

A. *Topes-enabled operating system clipboard*

1) *Study context*

In our first case study, we aimed to integrate topes into the Windows operating system clipboard, in order that users could copy a string from some application X in some format F_X and paste the string into another application Y in another format F_Y . This case study was motivated by our finding in prior user studies that office workers often needed to manually perform these transformations during their daily work when transferring data among web applications and spreadsheets [10]. We identified ten very common user tasks; of these, eight involved reformatting or extraction operations that could provide a useful basis for testing the usefulness of topes. The most common kinds of data transformed in tasks were dates, person names (e.g.: "Scaffidi, Chris"), locations (written as city and state, e.g. "Los Angeles, CA"), and phone numbers (Table 2). For example, spreadsheets in the Per Diem Lookup task normally formatted locations like "Los Angeles, CA", but using these on the web site required extracting the state name "CA" and reformatting the city name to capital letters.

Table 2. User tasks involving Reformatting or Extraction operations [10]. (Uppercase=frequent operation; lowercase=rare)

Task name (see [10] for details)	Date	Person name	Location	Phone number
Currency converter	R			
Package Tracker				
Path to Procurement				
Peoplesoft Scraper		r	r	
Per Diem Lookup	E		RE	
Person Locator Scraper	R		e	
Scraper for CMS	re			
Staff Lookup		re		R
Stock Analysis	RE			
Watcher for eBay	r			

2) *Software developed in this case study*

We enhanced the Windows operating system clipboard to invoke topes and transform strings when performing the tasks we observed. We implemented a new Windows service that runs in the background and waits for keystroke combinations. After a user copies a string from one application (e.g. with Control+C), the user switches to the other application (e.g.,

with Alt+Tab) as if about to paste. However, rather than pasting directly (e.g., with Control+V), the user presses F12, which launches a new window that we implemented (Figure 3). This window populates itself by examining the string on the clipboard, identifying the tope that best describes that string (with an algorithm described in [11]), and showing a list of strings that would result from performing the tope's operations.

When the user selects a format and clicks "Ok", the string is transformed accordingly and pasted into the target application. Moreover, the clipboard remembers what operation the user selected; if the user subsequently copies another string and types Control+F12, then that operation is replayed on the new string to produce a new value that is pasted. In this way, it is possible to reformat a series of strings (as we observed in user tasks) without having to manually perform reformatting or extraction operations.

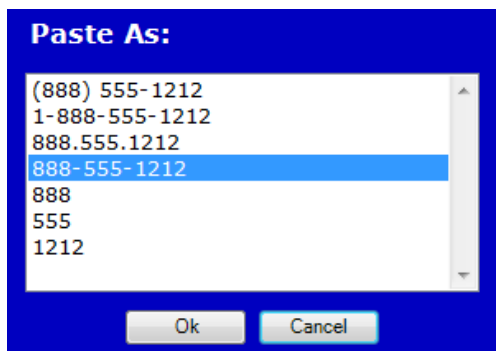


Figure 3. A Windows clipboard enhancement that uses topes to automate reformatting and extraction operations

3) Study findings

While writing code for the study over the course of 1 week, the most significant difficulties encountered were related to the Windows API. In particular, it was difficult to find the right combination and ordering of API calls to invoke in order to complete the paste operation (which was complicated because the popup window ran in a different process than the target application, and because focus had to be set to the correct window in the target application before the paste message could be sent). We found that invoking topes was simple because our new Windows service could be completely implemented in C#, which was compatible with the TDE interface for calling topes. For the four common kinds of that need to be reformatted in user tasks (Table 2), we had no difficulty in implementing and invoking the corresponding topes. Of these kinds of data, only dates can be easily transformed with existing C# APIs. Therefore, the code for reformatting the other three was greatly simplified by using topes and the TDE. In this study, the operating system clipboard acted as a repository, so we essentially applied topes in locus #2 to the repository architectural style.

B. Topes-enabled web macro tool

1) Study context

While Case Study A produced an enhanced operating system clipboard that simplified the copy-transform-paste process by automating the transformation step, we sought to automate the entire copy-transform-paste process by integrating topes into a scripting environment that could automate all three steps of the process. In particular, since many tasks involved interacting with web sites, we hoped that these tasks could be automated with a web macro recorder, which is a browser extension that records interactions with web sites and then plays them back [9]. To date, one of the most well-developed web macro tools is the CoScripter extension for Firefox. For example, a CoScripter web macro might specify that the browser should go to a particular web application X, copy some strings from the website, go to another web application Y, and paste the values into a web form on Y. But X and Y might use different formats for strings. Since CoScripter cannot fully automate this reformatting, the user must intervene.

Moreover, this can even be a problem when copying strings from the user's personal configuration file and pasting strings into a web application. For example, suppose a user wants to automatically register a phone number on the National Do Not Call list¹, where the web form requires that the phone number should be split into two fields (area code and local 7-digit number). In this case, the current version of CoScripter would require the user to manually enter these pieces of the phone number into two separate configuration file entries, which requires extra time, presents an extra opportunity for error, and represents potentially redundant work if the user already has a "phone" variable containing the string in its entirety.

2) Software developed in this case study

To evaluate the usefulness of topes for helping users to automate tasks, we have modified the CoScripter web macro tool to now support instructions specifying that reformatting or extraction operations should be executed on strings.

To support reformatting operations, the tool now supports instructions like *paste in "02/20/2009" format from "date" into the "travel date" textbox*. This instruction tells CoScripter to parse the string on the tool's internal clipboard with the "date" tope, to reformat the parsed string so that it is formatted like the example "02/20/2009", and then to paste the result into the textbox labeled "travel date". (Of course, the user must specify an example string that unambiguously identifies the desired format... "01/01/01" will not do.)

To support extraction operations, the tool also now supports instructions like *paste "state" in "CA" format from "location" into the "destination" textbox*. This instruction tells CoScripter to parse the string on the tool's internal clipboard with the "location" tope, to extract the part named "state", reformat that part so it looks like "CA", and to paste the result into the textbox labeled "destination".

¹ <https://www.donotcall.gov/register/reg.aspx>

To edit existing topes or to create custom new topes, users can access the TDE's tope editor (Figure 1) through a new menu item. By creating or customizing topes, then writing instructions like those above, it is now possible to more fully automate user tasks that we observed: users no longer need to pause the macro and manually do reformatting and extraction.

3) *Study findings*

The most significant problems encountered during this 1 month study were in finding where to modify CoScripter's complex, lightly-documented JavaScript source code, in order to invoke topes. The other significant obstacle encountered during this case study was that CoScripter cannot invoke topes directly through our TDE interface, since JavaScript in Firefox cannot call C#. Consequently, we wrote a Firefox extension in C++ that acted as a bridge between CoScripter's JavaScript code and the C# TDE interface for invoking topes.

As in Study A, the four common kinds of strings that need to be reformatted in user tasks (Table 2) could easily be expressed with topes. Of these kinds of data, only dates can be easily transformed with existing JavaScript APIs. If more than a few other kinds of data needed to be supported (including those shown in Table 2), then this would outweigh the additional complexity introduced by our C++ bridge. But if only a handful of kinds of data needed to be supported (such as just those shown in Table 2), then it probably would be preferable to simply code the requisite rules in JavaScript. The benefit of our more complex implementation, therefore, is that it opens up the possibility for users to create and invoke custom new kinds of topes for their web macros.

Referring back to the discussion from Section IV, CoScripter embodies the interpreter architectural style: the macro is the script S, CoScripter is the interpreter I, and the browser is the component T manipulated by the interpreter as it reads instructions from the script. We have essentially enhanced the I-T data transfer at locus #1 by modifying I so that it invokes a tope prior to pasting strings into T. (While CoScripter overall is an interpreter, its clipboard in particular acts as a repository; in this view, we applied #2 to a repository style, as in Study A.)

C. *Topes-enabled web services*

1) *Study context*

Prior to becoming researchers, each of us worked as a professional programmer for several years. During that time, we frequently needed to invoke web services from applications or from other web services. In some cases, we also needed to write programs that read data off of websites via http.

At present, XML is a popular representation for data sent to and from web services. XML is an improvement over earlier approaches (such as CORBA) that relied on binary serialization of data. The reason is that programmers can easily inspect XML emitted by an element (or even the XML's DTD or XSD specification, when one exists), making it straightforward to design an application that consumes the XML as well as to detect schema errors (when a DTD or XSD is available).

Another common data exchange mechanism is for software elements to read HTML from web pages, rather than from carefully designed XML streams. While this makes it possible to consume information that is currently not available in XML, the relatively unstructured nature of HTML requires elements to carefully sift through the HTML to find needed data.

Microformats are a compromise between the careful design of XML and the loose structure of HTML [7]. In this mechanism, when a software element emits HTML, it affixes a "class" attribute to HTML tags to specify a category for the tag's text. For example, a tag containing a phone number might carry "class=tel". Commonly recognized labels are published on a wiki so that other people can create new elements that download labeled HTML and retrieve data. (Technically, microformats are actually a simplified adaptation of the semantic web, which uses a more complex and heavyweight tag-labeling mechanism [2].)

Neither XML nor microformats address data heterogeneity. For example, one web service returns a list of holidays that will occur in a given month.² The dates returned are formatted like "2010-12-31T00:00:00.0000000-05:00". Actually using such a value typically requires getting the data out of the XML returned by the server, removing the time portion (which is irrelevant, anyway, for a holiday), and reformatting the date to a more user-friendly format.

Extra reformatting operations can also be necessary when sending data to a server. For example, the CDYNE411 reverse phone lookup service requires that phone numbers sent to the service must be formatted like "2024561111".³ Thus, applications that call the service must reformat strings from more common formats (like "202-456-1111") to the format required by the web service.

2) *Software developed in this case study*

We developed two kinds of software to test the usefulness of topes when transferring data in this setting.

First, we implemented a series of applications that each called a web service. In particular, we implemented applications that communicated with the "holiday" web service and the CDYNE411 web service, above. In addition, we implemented other applications that used services for looking up the birth and death dates of famous people, services for validating credit cards, and services for looking up a person's phone number.

Each application transformed and transmitted data in the format required by the web services, retrieved the results, and then transformed the data that was retrieved if needed. For example, the application that called the CDYNE web service would accept a phone number in any format, invoke a tope to reformat the string to remove the hyphens or other separators, transmit the new string to the web service, and extract the phone number registrant from the result value.

² <http://www.holidaywebservice.com/Holidays/US/USHolidayService.aspx>

³ <http://ws.cdyne.com/cdyne411ws/cdyne411.aspx>

```
.tel { tope:url(http://myserver.com/phones.xml); }  
.date { tope:url(http://www.w3c.org/date.xml); }
```

Figure 4a. Example of a topesheet, mapping from microformat labels (e.g., “tel”) to URLs where appropriate topes are stored (serialized as XML).

```
String html = ... retrieve from web page as usual  
String turl = ... URL of topesheet, above  
ItemLoaderForHtml loader = new ItemLoaderForHtml(html, turl);  
ItemSet items = loader.Load(".tel");  
List<String> tels = items.FormatAs("(888) 555-3030");
```

Figure 4b. Code for retrieving phone numbers from a web page and putting them into a desired format.

The second kind of software implemented for this field study was a new C# library that uses topes in retrieving and transforming strings from microformat-labeled HTML. There are two steps for using this library. First, the programmer would create a “topesheet”, which provides a mapping between microformat labels and topes (Figure 5a). These topes could have been created by the programmer, or they might be “standard” topes provided by a standards body such as the W3C. (We chose this topesheet syntax for consistency with the CSS syntax already used for HTML.) Second, the programmer would invoke our library to actually read strings from HTML and reformat them with topes (Figure 5b). For example, the programmer could use the library to retrieve all of the phone numbers from HTML and transform them to a particular format.

Our library supports several options. First, the HTML itself is allowed to specify a topesheet, meaning that the provider of the HTML can supply this metadata in order to help applications to extract and transform data. The topesheet in the HTML is used as a default, if the programmer of the client application chooses not to override it with a custom topesheet. Second, our library can be used to retrieve and reformat strings from XML (rather than HTML). In this case, the desired elements are referenced using XPATH notation rather than microformats. Finally, the programmer can specify extra parameters in order to filter strings based on how well they match the tope’s constraints. (By default, the library discards strings that violate constraints that should “always” or “almost always” be true, or that violate several “often” constraints.) If the data consumer finds that insufficient data meets these criteria, then it can take an appropriate action such as logging an error, sending an email to a system administrator, or failing over by connecting to an alternate data provider.

3) Study findings

During the 2 months spent on this study, the only major problem encountered was that many of the web services were unreliable and often offline. We did not encounter any major problem related to topes at all. In effect, our library demonstrates a successful use of topes at locus #1 (before transfer of inputs to web services and sites) as well as at locus #3 (after transfer of outputs from web services and sites).

Upon reflection on this study, we realized that topes and our library could be used to help insulate data consumers from unannounced format changes on web services and web pages. For example, a web page might produce phone numbers in some format F_A at one point in time, and an application might use our library to transform these strings to a format F_Z . But at some later point in time, the data provider might be modified so that it produces strings in format F_B rather than F_A . These new strings could be handled *automatically* with *no modifications to code* by the client, as long as the tope contains formats F_A , F_B , and F_Z . The tope would automatically detect that the incoming strings are in format F_B rather than F_A , and it would transform them to F_Z accordingly. If the topesheet is specified by the data producer, rather than the data consumer, then the data producer could update the tope if needed so that it is sure to contain all the necessary formats. In other words, topes and our library make it possible for a data provider to *completely insulate* consumers from changes in data formats, at least at the level of individual string values (data heterogeneity).

VII. DISCUSSION AND FUTURE WORK

This paper has explained how topes can be used to automate reformatting and extraction operations often involved in transferring strings between applications. In particular, casting the problem in abstract architectural terms enabled us to identify three loci for using topes to solve this problem. Of these loci, two are applicable in a range of architectural styles.

Using three case studies, we evaluated four propositions, leading to the following findings.

First, we found that most of the difficulties encountered in the studies indeed resulted from technologies other than topes or the TDE. These problems included difficulty understanding the operating system API and difficulty understanding the code of the CoScripter web macro tool. In order to invoke topes from JavaScript in CoScripter, we had to write a C++ bridge that added extra complexity. These difficulties are not particularly unique to topes but rather illustrate the challenges of integrating any new functionality into an existing application or framework.

Second, we found that topes were indeed able to perform the string transformations needed in a variety of situations. In particular, Case Studies A and B were motivated by prior empirical studies that identified several key kinds of strings that frequently needed to be transformed, and topes were able to handle all of these data without difficulty.

Third, we found that topes were indeed useful at all three loci (before/during/after data transfer), though not necessarily in every combination of locus and architectural style. In particular, locus #2 (transformation during data transfer) was only useful for doing transformations when a repository served as a connector between two components. In general, loci #1 and #3 (before and after transfer) were more useful because, as we expected, there is more of an opportunity to modify to components than connectors.

Fourth, we found that using topes did generally simplify the code required to perform string transformations. The only exception to this rule was during the CoScripter study, where integrating into the application required writing a C++ bridge. Together, these three studies highlight the usefulness of topes as long as massive amounts of new bridge code are not required.

Based on the results of this work, we could now build on topes to support larger transformations. Many techniques exist for handling schema heterogeneity in specific settings (Section II), but a generalized mechanism has been lacking for specifying setting-independent transformations of individual strings in structures' "leaf nodes." Topes fill this gap, so integrating topes with structural techniques would provide a complete top-to-bottom solution for structural transformation and reusable string-level transformation. This integration would make it possible to perform large, increasingly complicated tasks that transfer data between application, without laborious manual effort or having to write complex, messy code.

REFERENCES

- [1] D. Abadi, et al. Scalable semantic web data management using vertical partitioning. *Intl. Conf. Very Large Data Bases*, 2007, 411-422.
- [2] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284, 5 (May 2001), 34-43.
- [3] A. Bonifati, et al. HePToX: Marrying XML and heterogeneity in your P2P databases. *Intl. Conf. Very Large Data Bases*, 2005, 1267-1270.
- [4] D. Dotan and R. Pinter. HyperFlow: An integrated visual query and dataflow language for end-user information analysis. *Symp. Visual Lang. and Human-Centric Computing*, 2005, 27-34.
- [5] F. Hakimpour and A. Geppert. Resolving semantic heterogeneity in schema integration. *Intl. Conf. Formal Ontology in Info. Sys.*, 2001, 297-308.
- [6] A. Halevy. Why your data won't mix. *ACM Queue*, 3, 8, 2005, 50-58.
- [7] R. Khare and T. Çelik. Microformats: A Pragmatic Path to the Semantic Web. *WWW'06: Proc. 15th Intl. World Wide Web Conf.*, 2006, 865-866.
- [8] W. Kim and J. Seo. Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer*, 24, 12, 1991, 12-18.
- [9] G. Leshed, et al. CoScripter: Automating & sharing how-to knowledge in the enterprise. *Conf. on Human Factors in Computing Systems*, 2008, 1719-1728.
- [10] C. Scaffidi, et al. Using scenario-based requirements to direct research on web macro tools. *J. Vis. Lang. and Computing*, 19, 4, 2008, 485-498.
- [11] C. Scaffidi, B. Myers, and M. Shaw. Intelligently creating and recommending reusable reformatting rules. *Intl. Conf. Intelligent User Interfaces*, 2009, 297-306.
- [12] C. Scaffidi and M. Shaw. Accommodating data heterogeneity in ULS systems. *Intl. Workshop Ultra-Large-Scale Software-Intensive Sys.*, 2008, 15-18.
- [13] J. Stylos, B. Myers, and A. Faulring. Citrine: Providing intelligent copy-and-paste. *Symp. User Interface Software and Technology*, 2004, 185-188.
- [14] R. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*, Wiley, 2009
- [15] R. Yin. *Case Study Research: Design and Methods*, 3rd Edition, Sage Publications, 2003
- [16] C. Zheng, W. Ou, Y. Zheng, and D. Han. Efficient Web Service Composition and Intelligent Search Based on Relational Database. *International Conference on Information Science and Applications (ICISA)*, 2010, 1-8.