Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), at the International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 2008, pp. 15-18

Accommodating Data Heterogeneity in ULS Systems

Christopher Scaffidi
Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
cscaffid@cs.cmu.edu

Mary Shaw
Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
mary.shaw@cs.cmu.edu

ABSTRACT

Ultra-Large Scale (ULS) systems comprise numerous software elements designed and implemented by independent stakeholders whose requirements may vary widely. Consequently, elements in a ULS system may use different data formats, which complicates integration of elements. Writing code to robustly convert data from one format to another requires time and skills that some programmers may lack. Worse, the stakeholders who control a software element may change the element's data format at any point in the future without warning, causing format incompatibility not foreseen during the ULS system's construction.

To address heterogeneity of data formats, we present a new abstraction called "topes". Each tope describes one kind of data, including known formats of that data and rules for transforming values among formats. Labeling the inputs and outputs of software elements with topes raises the level of abstraction so that elements produce and consume certain kinds of data, rather than particular formats.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability – *Data mapping*

General Terms

Reliability, Languages.

1. INTRODUCTION

To support data aggregation or analysis goals, many organizations such as government agencies and large corporations have already begun to construct Ultra-Large Scale (ULS) systems by connecting hundreds or thousands of elements. These elements include software applications, servers, programming platforms and other software components. In many cases, independent groups of stakeholders initially designed these elements with little or no awareness that the elements would later compose a part of a larger ULS system [10]. While professional software organizations perform most software integration, ordinary end users also may perform some integration in a ULS system by "mashing-up" data from multiple elements. A natural result of this decentralization is an assortment of independently evolving, heterogeneous elements to be somehow stitched together into a ULS system.

In particular, a task in a ULS system may call for integrating two existing elements by exchanging data between them, but the two may require the data to be in different formats. For example, one element may write mailing addresses with a full street type such as "Avenue" while another may use abbreviations such as "Ave.", and one may reference books by ISBN while another may use book titles. Since ULS systems are often embedded in a social context, data may also come from users, who are certainly not known for consistency in their formatting of data.

Thus, constructing a ULS system often requires connecting elements to data sources through code that transforms data from one format to another, and creating this code demands additional time and effort. Yet even this does not guarantee proper behavior, since software elements evolve independently and (like users) may make unannounced changes in data format any time in the future.

In order to detect that a data source has begun using a different data format, other elements could attempt to validate the data. One obstacle to thorough validation is that the two commonly practiced validation approaches, numeric constraints and regular expressions ("regexps") [3], are "binary" in that they only attempt to differentiate between definitely valid and definitely invalid inputs. Yet for many kinds of data such as person names and book titles, it is difficult to conclusively determine validity. For instance, no numeric constraint or regexp can definitively distinguish whether a string is a valid person name.

A more effective validation approach should not only identify definitely valid and invalid data when feasible, but also identify questionable inputs—values that are not clearly valid but are also not clearly invalid—so those data can receive additional checking from people or programs to ascertain validity. For example, if a web service provides a questionable person name that has an odd mix of uppercase and lowercase letters, such as "LincoInshire MCC", then another software element could log the value so a system administrator could double-check it. Or the software element might call a different web service to double-check the value. Each such strategy first requires identifying questionable values.

Based on these considerations, we provide a new abstraction called "topes" to promote data interoperability without requiring elements to use a common data format. Each tope describes one kind of data (such as a mailing address) by providing functions to recognize and transform data among the formats of that kind of data. In addition, each tope contains functions for detecting when data values might be invalid, thereby enabling software elements to detect when a data provider might be malfunctioning at runtime.

Section 2 outlines the current state of data exchange in ULS systems. Section 3 introduces topes and describes how they enable a ULS system to accommodate data heterogeneity. Section 4 discusses reusability of topes. Section 5 summarizes how topes help accommodate data heterogeneity in ULS systems and concludes by discussing future work.

ULSSIS'08, May 10–11, 2008, Leipzig, Germany.

2. DATA EXCHANGE MECHANISMS

At present, XML exchange is a popular mechanism for inter-system communication, finding use in service oriented architectures, web services and RSS feeds. In terms of creating ULS systems, 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 (or even the DTD specification, when one exists), making it straightforward to design a new element that consumes the XML. Organizations can then incrementally accumulate elements and develop a ULS system.

Another common data exchange mechanism is for software elements to read HTML from web pages, rather than from carefully designed XML streams. Such elements include search engines, product review aggregators [7], and web macro recorders [9]. While this makes it possible to consume information that is currently not available in XML, HTML's relatively unstructured nature requires elements to carefully sift through the HTML to find needed data. In response, researchers are developing various information extraction algorithms that find certain kinds of data in HTML (such as dates, company names, and person names) [5].

Microformats are a compromise between the careful design of XML and the loose structure of HTML [8]. 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. This obviates the need for sophisticated and brittle information extraction algorithms, retains human-readability, and requires minimal effort to retrofit existing HTML-producing elements so that they label their output with category names. (Microformats are actually a simplified adaptation of concepts in the semantic web, which uses a more complex and heavyweight tag-labeling mechanism [2].)

Unfortunately, none of the mechanisms outlined above address data heterogeneity. That is, XML and microformats enable a data consumer to find the correct fields, but these fields may be in the wrong format, requiring a programmer to write transformation code. While XSL can be used to reorder XML nodes, it cannot reformat a phone number, such as from ###-###-#### to (###) ###-####. A programmer could write code that performs this transformation, but the data-producing element could begin using a different format (or begin malfunctioning) at any point in the future, causing the data-consuming element to malfunction. It is also common for XML feeds to be syntactically correct (conforming to a DTD) yet to contain questionable numbers [11], which could also cause the XML consumer to malfunction.

3. TOPES AND HETEROGENEITY OF DATA

Our approach models each kind of data as an abstraction called a "tope", which contains functions for recognizing and transforming one kind of data between formats [13]. For example, a tope for email addresses might have a format that recognizes a username, followed by an @ symbol and a hostname. At the simplest level, the username and hostname could contain alphanumeric characters, periods, underscores, and certain other characters. Formats can reject a string, accept a string, or return a number between 0 and 1 to indicate confidence in the string's validity. For example, an email address with 64 characters in the username would technically be valid but probably questionable.

Multiple patterns are necessary for describing kinds of data that may appear in more than one format. For example, companies can be referenced by common name, formal name, or stock ticker symbol. Common names are typically one to three words, sometimes containing apostrophes, ampersands, or hyphens. Formal names may be somewhat longer, though rarely more than 100 characters, and they sometimes contain periods, commas, and certain other characters. Ticker symbols are drawn from a finite set of officially registered symbols. These three formats *together* comprise a tope describing how to describe company names. The tope would include a lookup table for transforming among these three formats.

Implementing topes

Just as an abstract type is not executable, topes are not directly executable but must be implemented. For some simple single-format topes, programmers could use a regexp to implement a format. However, regexps are insufficient for identifying questionable values or transforming strings, so we have provided a Tope Development Environment (TDE) supporting more sophisticated mechanisms for implementing topes [12].

To implement a tope format, a programmer uses the TDE to describe the data as a sequence of named parts, with facts specified about the parts (Figure 1). Facts can be specified as "always", "almost always", "often", or "never" true. The user interface has an advanced mode where programmers can enter facts that "link" parts, such as the intricate rules for validating dates.

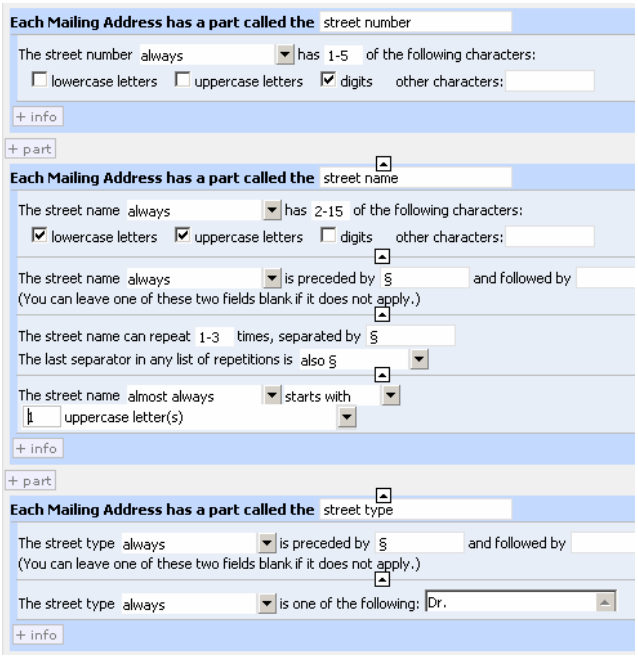

Figure 1: Describing street addresses. § indicates "space".

Again using an editor based on sentence-like prompts, a programmer implements each transformation as a series of instructions that can change separators, reorder parts, use lookup tables on parts (such as changing "Dr." to "Drive" in a street address), change capitalization of parts, and call other transformations as functions on parts that match another tope.

The TDE saves the tope in an XML file, which now contains an abstract description of one kind of data, independent of any particular application, software element, or application platform.

Using topes

Using a tope involves two steps. First, elements that produce data label each field with a certain tope (much as programmers label variables with types in traditional source code). Second, software elements read labeled data fields through an API that we have provided. At runtime, the API uses the tope's formats to verify that each field contains valid data, then uses the tope's transformations to reformat the data into the format needed by the data consumer.

These two steps are performed slightly differently depending on whether the data are exchanged via HTML, XML, or some other data source, but the same topes can be reused without modification.

As a concrete example, suppose that a software element produces HTML with some tags that contain phone numbers. These tags would be labeled with class="tel" (as in microformats), and the HTML would also contain a topesheet reference:

```
<!-- topesheet=http://myserver.com/mytopes.ts -->
```

This topesheet reference tells any consumer of the HTML where to find a list of tope implementations for the tags in the HTML. The topesheet at this URL would contain code like the following:

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

This topesheet indicates that tags labeled as "tel" can be validated and transformed with the tope implementation located at the specified URL. Universal topes such as dates could be shared by many people, but developers could publish topes for organization-specific kinds of data, such as project codes. (We chose this topesheet syntax for consistency with the CSS syntax already used for HTML.)

The second step of using topes requires the data consumer to read data fields through our C# API. For example, the following code reads phone numbers from the HTML and puts them into a uniform format:

```
ItemLoader loader = ItemLoader.FromHtml(html);
ItemSet items = loader.Load(".tel");
List<String> tels = items.FormatAs("(888) 555-3030");
```

In the first line, the ItemLoader loads topesheets and topes referenced by the HTML. For each format in each tope, the ItemLoader creates a context-free grammar and attaches the format's constraints (specified in the TDE) to the relevant productions of the grammar.

In the second line of code, the ItemLoader retrieves each microformat-labeled tag and parses the text against the referenced tope's format grammars. The ItemLoader discards strings that fail this parse or that violate too many constraints (discussed below).

In the third line of code, the ItemSet determines which of the tope's formats best matches the specified prototype and then uses the tope's transformations to reformat each string to the format. Note that the original format of the phone numbers is irrelevant—phone numbers could be written as "+1 888 555 3030", "888-555-3030" or any other format included in the tope, or even a mixture of those formats. In any such case, the code above yields a consistent list of strings in the format required by the data consumer. (If some values could match more than one format, then the programmer obviously should specify an unambiguous prototype, such as the date "04/31/99" rather than "01/01/01".)

This approach helps insulate the HTML consumer from unannounced format changes. For example, the producer might suddenly write data in one of the tope's other formats, and the consumer would automatically recognize and transform these to the needed format. Note that consumer code never refers to a tope by URL or to a format by name.

Moreover, in the second and third lines of code, the programmer can pass extra parameters to filter strings based on how well they match the grammar's constraints. (The default, shown above, 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 based on how questionable the data are, such as logging an error, sending an email to a system administrator, or failing over by connecting to an alternate data provider.

The topes provided by data producers might not contain a format required by the data consumer. In that case, the programmer of the data consumer can download the topes and add the requisite formats. A custom topesheet can then be passed into the ItemLoader constructor to override the topesheet provided by the producer. Caching the producer's topes (without customization) may also be helpful for reverting the topes in case the producer makes unhelpful changes to the topes at some point in the future.

All of these features are also available for XML data—though in this case, tags are referenced with XPATH notation rather than microformats. Indeed, the same topes can be used without modification for HTML and XML data, as well as spreadsheet data [13] and (with a suitable API) data sources that are not yet invented.

4. REUSE OF TOPES

In order to simplify the process of finding useful topes, the TDE will eventually include a repository system where programmers can publish and share topes. Because not all programmers have the same skills and goals, different tope implementations will have different quality and semantics (even when they are intended to describe the same kind of data). Consequently, we are developing techniques to help programmers find and select topes that are appropriate for a particular use.

The role of evidence in software reuse

When considering whether to reuse a component, tope or any other piece of software, a programmer must rely on evidence about the software in order to determine whether it has suitable functionality and quality attributes to satisfy a particular reuse case. Evidence comes from different sources, including four that have gained widespread acceptance among computer scientists [14]: formal verification, code generation by a trusted automatic generator, systematic testing, and careful empirical studies of the software in operation.

We consider these "high-ceremony" sources of evidence because, like high-ceremony software development processes [4], these require precise specifications and substantial investment of effort. Consequently, in practice, high-ceremony sources of evidence are often unavailable, so programmers instead generally rely on "low-ceremony" evidence from other sources, such as:

- Informal human-readable documentation
- Advertising claims by vendors
- Experiences of co-workers
- Informal ratings and comments posted in online forums
- Formal product reviews (in professional journals or on web sites)
- Careful certification by product testing laboratories

Two low-ceremony sources of evidence, reputation and references, have been used in repositories to help programmers select Matlab functions [6], web services [15], and other software components [1].

While this prior work has established the feasibility of using certain low-ceremony evidence to facilitate code reuse, several questions remain unexplored. What other forms of low-ceremony evidence could be used to guide programmers to software with appropriate functionality and quality? Which forms of low-ceremony evidence are most beneficial, and under what conditions? What analyses can we perform to distinguish credible evidence from less credible evidence? What other analyses and user interfaces can we provide to improve the usefulness of this evidence? Tope reuse affords us the opportunity to explore these and other questions.

Low-ceremony evidence and tope sharing

When a programmer needs a tope for a certain kind of data, the key functional and quality attributes are:

- Functional relevance: Does the tope describe this kind of data?
- Correctness: Does the tope accept/reject/question the *right* strings (as defined by the person who would reuse the tope)?
- Consistency: If one format accepts/rejects/questions a string, and the string is transformed to another format, does the second format also accept/reject/question the transformed string?

To capture evidence about functional relevance, we will provide tools enabling programmers to annotate topes with a human-readable name, as well as a textual description that essentially serves as documentation or advertising claims by the programmer. Further evidence will come from the filename in the URL of the tope (as in "http://.../phones.xml") as well as the microformats ("tel") and XPATH ("/USER/PHONE") that associate fields with the tope.

To capture evidence about correctness, our tools will allow programmers to log and publish example strings that are accepted, rejected, or questioned by a tope. Moreover, other examples could be published by other people (such as co-workers, other programmers, formal product reviewers and engineers at certification labs) along with a rating of the tope's quality.

To generate evidence about the internal consistency of topes, we will design an automated testing tool that passes the example strings into the formats and transformation functions, then checks for non-contradictory responses. (We are also developing an improved version of the TDE that will help prevent many of the most likely forms of inconsistency.)

Finally, we will provide tools enabling programmers to search for topes that will yield high benefits (in the form of reliable operation) and few customization costs (in the form of customization and debugging effort). Search results will rank topes based on a combination of functional relevance, correctness, and consistency. Our TDE will record how much time is spent customizing and debugging topes selected by the programmer. We then will feed these results back into the ranking algorithm so that it better reflects the relationship that links the available evidence to functional relevance, correctness, and consistency.

5. DISCUSSION AND CONCLUSION

Traditionally, software elements have required the data that they consume to be in particular formats. Topes raise the level of abstraction so that elements require certain kinds of data, rather than particular formats. For instance, an element now can consume phone numbers, rather than phone numbers in a certain format. In effect, we enable an element's abstract type or interface to call for inputs with certain *semantics*, rather than certain syntax. Decoupling the formats of information providers and information consumers not only simplifies the construction of ULS systems from elements with heterogeneous data formats, but it also insulates against future format changes.

Moreover, tope formats can identify questionable strings that might be invalid. This makes it possible for elements to detect when it might be appropriate to trigger fail-over techniques so that an element in a ULS system can double-check data provided by other elements or users.

Our future work in developing a tope repository will enable us to collect actual tope implementations as well as feedback from people using topes in real applications. This will facilitate incremental TDE improvements to further assist programmers as they implement and reuse topes to validate data in ULS systems.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] Bellettini, C., et al. User Opinions and Rewards in Reuse-Based Development System. *Proc. 1999 Symp. Software Reusability*, 1999, 151-158.

[2] Berners-Lee, T., Hendler, J., and Lassila, O. The Semantic Web. *Scientific American, 284*, 5 (May 2001), 34-43.

[3] Bhasin, H. *Asp.NET Professional Projects*, 2002.

[4] Booch, G. Developing the Future. *Comm. ACM*, *44*, 3 (2001), 118-121.

[5] Chakrabarti, S. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann, 2002.

[6] Gulley, N. Improving the Quality of Contributed Software and the MATLAB File Exchange, *2nd Workshop on End User Software Engineering*, 2006, 8-9.

[7] Hu, M., Liu, B. Mining and Summarizing Customer Reviews. *Proc. 10th SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2004, 168-177.

[8] Khare, R., and Çelik, T. Microformats: A Pragmatic Path to the Semantic Web. *Proc. 15th Intl. World Wide Web Conf.*, 2006, 865-866.

[9] Little, G., et al. Koala: Capture, Share, Automate, and Personalize Business Processes on the Web. *Proc. Conf. on Human Factors in Computing Systems*, 2007, 943-946.

[10] Northrup, L. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, 2006.

[11] Raz, O., Koopman, P., and Shaw, M. Semantic Anomaly Detection in Online Data Sources. *Proc. 24th Intl. Conf. Software Engineering*, 2002, 302-312.

[12] Scaffidi, C., Myers, B., and Shaw, M. Tool Support for Data Validation by End-User Programmers, Research Demo for *Proc 30th Intl. Conf. Software Engineering*, 2008, to appear.

[13] Scaffidi, C., Myers, B., and Shaw, M. Topes: Reusable Abstractions for Validating Data, *Proc 30th Intl. Conf. Software Engineering*, 2008, to appear.

[14] Shaw, M. Writing Good Software Engineering Research Papers. *Proc. 25th Intl. Conf. Soft. Eng.*, 2003, 726-736.

[15] Zeng, L., et al. QoSAware Middleware for Web Services Composition. *IEEE Trans. Software Engineering, 30*, 5 (May 2004), 311-327.