Developing XML Vocabularies for Web Services

Calvin Smith
Patrick Garvey
Robert Glushko

School of Information Management & Systems
University of California, Berkeley
{calvins,pgarvey,glushko}@sims.berkeley.edu

4 January 2003

Center for Document Engineering Technical Report 2003-1

Abstract

Web services depend on XML documents that can be understood by the parties (or applications) that wish to do business, which requires that they agree on the meaning and organization of the information elements the documents contain. Without an agreement on an XML vocabulary, the glib promises of "easy integration" and "interoperability" with web services are undermined by the need for custom coding to transform instances of one side's data model into the other's. In this paper we discuss methodological and design issues that emerged in our work with the Talaris Services Business Language™ (SBL), an XML vocabulary developed to make it easier for suppliers of business services to integrate with a Talaris application that is used by corporate employees to purchase business services on the Web. Talaris chose to use the emerging Universal Business Language (UBL), a "horizontal" XML vocabulary of common semantic components, as a basis for SBL to give it more robust semantics and to encourage its adoption.

Web Services and XML

Web services are being hailed as a breakthrough in exploiting the Internet as an application development and integration platform. Applications that conform to web services standards will "easily integrate with other services" and their "interoperability allows businesses to … more easily create innovative products, business processes and value chains" [IEEE 2002]. Similarly, XML – the Extensible Markup Language – is being touted as the *lingua franca* of the computer-processable Internet. HTML is not expressive enough, and EDI not flexible enough to handle inter-application communication between a wide-range of ever-changing companies and services. XML's formality and flexibility to describe arbitrary data formats overcomes the limitations of the fixed tag set of HTML and the syntactic rigidity of traditional EDI for electronic business.

Nevertheless, these claims about the benefits of XML and web services are overly simplistic and at best only partly true. Web services standards enable businesses to present application interfaces to each other that hide details of their implementation. Web services standards also ensure that the XML documents that invoke services or return their responses can be reliably sent and received in the appropriate sequence. But these are only syntactic guarantees. Nothing in any web service standard and nothing about XML *per se* ensures that the XML documents exchanged by web services are both meaningful and mutually intelligible. This requirement for well-defined semantics in web services trumps all other considerations; it does no good for a business to expose an application interface and agree to accept an XML document that embodies it if no other business can be sure it understands it or the XML document that it gets back.

So while XML's flexibility to describe documents of arbitrary content is its greatest strength, it is also its greatest weakness. Since XML has no fixed semantics, it can be used to describe anything. But this means that XML doesn't come "out of the box" with a standard way to describe anything. Using the same tags isn't sufficient, either. The same content will invariably be described using different names (a "*Price*" element in one document might mean the same as "*RetailPrice*" in another document), and different content might be given the same name ("*Price*" might mean "*Retail Price*" in one document and "*Wholesale Price*" in another). One can only hope that any business that offers a web service will unambiguously explain the meanings of the tags contained in the XML documents it sends and receives.

Standard XML Vocabularies

Web services and electronic commerce in general depend on documents that can be understood by the parties (or applications) that wish to do business, which requires that they agree on the meaning and organization of the information elements the documents contain. With EDI, dominant buyers or sellers have long imposed document requirements on anyone wanting their business (for example [HDI 2003]). Likewise, a pair of business partners might agree on the XML documents that are the inputs and outputs of their respective services, but this two-at-a-time approach doesn't scale and undermines the feasibility of partner discovery and "plug-and-play" implementation that are the core of the web services vision.

Efforts to create industry standard XML vocabularies are a significant step toward semantic interoperability with scalability. Vertical industries can have very rich content and specialized processes, which imply highly specialized document models. Thus there are substantial benefits when the companies in a particular industry or market share XML definitions. These include reduced development and maintenance costs and the elimination of custom "mapping" between the information models embodied in a company's business systems and those of its trading partners. Without vocabulary standards, the glib promises of "easy integration" and "interoperability" are undermined by the need for custom coding to transform instances of one side's data model into the other's.

Standard XML vocabularies are most often developed in specific vertical industries by trade associations or industry consortia, such as the Open Travel Alliance's efforts in the travel industry, or HL7's work in the healthcare industry. Sometimes vertical standards are initially proposed by an innovative company seeking to encourage other firms to join its "Internet community" by reducing their costs of doing business with it (see the registry at XML.ORG for where dozens of companies and industry groups make their XML definitions available to others [XMLO 2003]).

The set of legal elements and attributes in an XML vocabulary and the rules by which they combine are formally specified in an XML schema. Several schema languages for XML are in common use: Document Type Definitions (DTDs) are a legacy from SGML and are prevalent in publishing applications, and W3C XSD is taking hold for data-intensive or transactional types of document [VDV 2002]. A schema might state that a document must contain one *Person* element, and that a *Person* must contain a *Name* and an *Age*. A validating XML parser can then check XML instance documents against their specified schema document, and verify that the XML document is valid. In the case described above, the parser would raise an error if a *Person* contains a *Time* element or does not contain *Name* and *Age* elements. While DTDs treat most content as "just text," other schema languages can enforce stronger datatype restrictions, such as specifying that an *Age* element must contain integer rather than floating-point values.

Standard "Horizontal" Vocabularies

Each new XML specification for a particular industry or marketplace is a step forward for that community, but it proliferates definitions of information models that are common to many of them. Since any large company will sell products in both direct and indirect markets, maintain a supply chain for its direct inputs to its manufacturing processes, procure large amounts of indirect goods for its operations, post job offers in employment marketplaces, and so on, it is inevitable that document standards developed separately in each of these industries or contexts will be incompatible.

For example, descriptions of businesses and individuals, basic item details, measurements, date and time, location, country codes, currencies, business classification codes, and similar “atomic” or “primitive” information units are needed in every industry in a wide variety of documents. Reusing these semantic building blocks is essential to facilitating interoperability and efficient implementation across vertical standards and between the different steps in a business process, such as procurement, where much information is repeated between catalogues, purchase orders, shipping notices, invoices, and payments. Without common semantic definitions in the documents they exchange, web services cannot be easily combined or choreographed to support inter-enterprise business processes or value chains.

The XML Common Business Library

The oldest attempt to attack the problem of interoperability among vertical XML commerce applications with a horizontal library is the XML Common Business Library [XCBL 2003] on which work first began in 1997 and which is used by many of the largest B2B marketplaces. XCBL, developed by Commerce One, SAP, and other B2B solution vendors and marketplace operators, includes a set of reusable XML components that are common to many business domains, along with a set of document frameworks for creating documents with a common architecture. The dozens of common B2B documents built according to the xCBL frameworks can be understood from their common message elements and extended in predictable ways.

But because of the enormous implications that horizontal XML content standards have for technical interoperability and market acceptance, vendor-driven vocabularies like xCBL have been unable to become de facto standards. It is only through a broad-based and credibly open process that a universal set of semantic building blocks and basic documents can emerge.

The Universal Business Language

The most promising effort to create standard XML business documents and reusable components is called the Universal Business Language [UBL 2003] begun in 2001 with a targeted completion in 2003. UBL hastened its development by taking xCBL as a starting point and modifying it to incorporate the best features of other existing XML horizontal and vertical business libraries. UBL attracted a critical mass of participants from all over the world, including top XML and EDI architects from e-commerce vendors, XML technology firms, and a broad range of governmental and business organizations.

UBL uses a three-layered architecture to organize its reusable XML content and structures. At the top level, there is a schema and namespace that defines what UBL refers to as a functional area, such as *Order* or *Invoice*. Supporting each functional area, there is a schema that contains aggregate types, such as *AddressType* and *PartyType*. Finally, there is a schema for common leaf types, which are types such as *TextType*, *IdentifierType*, and *QuantityType* that are the components that are reused across multiple functional areas and the multiple common aggregate type libraries.

Case Study: The Talaris Services Business Language™

The heart of this paper is a case study describing the development of an XML vocabulary to support the procurement of Web-based services. The need for this vocabulary, called the Talaris Services Business Language or SBL [TAL 2003], was perceived early in 2002 by an innovative company called Talaris to make it easier for suppliers of business services to integrate with its Web application used by corporate employees to purchase those services on-line. For example, Talaris customers can schedule the shipment of packages or arrange web conferences, services that are offered on the Web by providers such as FedEx and WebEx.

As challenging as it is to develop an XML vocabulary of practical scale, it is even more difficult when reuse of another XML vocabulary is a design goal. But Talaris chose to use UBL as the basis for SBL because it offered several important advantages:

- Talaris can use UBL's definitions for common business components, giving SBL a stronger semantic foundation than if it defined redundant components
- By implementing with UBL, Talaris aligns itself closely to an important global standards effort, which will make it easier for other parties to justify adopting SBL
- Using UBL makes it easier to do mapping between SBL documents and other UBL based document types. This encourages interoperability even with non-SBL users.

In the remainder of this paper we discuss the design of SBL and our use of UBL as a semantic foundation within it. We hope our work contributes to the new field of "Document Engineering" that is evolving as a new discipline for specifying, designing, and implementing the electronic documents that request or provide interfaces to business processes via Web-based services [GLUS 2002]. The essence of Document Engineering is the analysis and design methods that yield formal models to describe the information these processes or services require.

Because of our focus on methodology, we consider specific features of SBL or UBL only to illustrate the methodological and design issues for developers facing similar challenges. We also de-emphasize specifics about SBL and UBL because they are unlikely to remain accurate; UBL has evolved since we conducted this work in the summer of 2002 using a "snapshot" of UBL, partly as a result of our experiences and problems we faced in using it. SBL likewise will substantially change as Talaris expands its coverage and robustness to support additional domains of service procurement.

The Structure of SBL

The first task that we engaged in after defining the problem space of SBL and deciding that we were going to use UBL was determining the high-level structure of SBL – that is, how many namespaces should we divide the domain into and how should we organize them. (Namespaces enable an XML application to use elements from numerous vocabularies by pre-pending a vocabulary identifier to their names). In this regard, we followed closely the current UBL approach to namespaces. The SBL architecture we adopted was a similarly layered architecture built on top of the UBL common leaf and aggregate types.

The distinctive characteristics of services procurement strongly shaped our architecture and design. SBL encompasses multiple verticals, such as package shipment, web conferencing, and airline flight procurement. Each of these verticals consists of multiple providers that provide the same or similar services, such as the way in which FedEx, UPS, DHL provide comparable package shipping services. Sometimes the providers are vendors, as with package shipping, and they supply similar services to those of other providers in the same vertical. But sometimes the providers may be brokers, as with brokers who sell airline flight tickets, in which case multiple providers may be selling exactly the same service – that is, multiple brokers could provide the very same seat on a particular flight. Another critical feature of services procurement is that some services are highly abstract and thus are described entirely by abstract data. For example, to schedule web conferencing there are very few “real” things that need to be described like packages and places. Components within such verticals will contain very little fixed or essential data.

Taken together, these characteristics of services procurement suggest that industry verticals (or groups of firms that provide the same services) form the appropriate boundaries for breaking down the SBL vocabulary into sub-vocabularies. Within each vertical industry area, there are many commonalities in functionality, and thus there should be significant reuse of components. On the other hand, the differences between verticals are substantial, and the extent of horizontal reuse is likely to be less than with the procurement of more tangible goods.

Nevertheless, after decomposing SBL into verticals, we have been able to identify some overlap between verticals. A component, such as *TimeAddressSet*, which specifies an address and a time window associated with that address, could be used across multiple verticals. For instance, in Package Shipment, it is used when specifying when and where to schedule the pickup of packages, but when renting a car, it could be used to specify a time range during which and a location to which a car rental will be returned. Therefore, we created a library of SBL Core Components that could be used across verticals.

The next decision was whether to break the vertical up into smaller pieces. We divided each vertical into separate schemas, one for each of the various classes of documents that the vertical contained. For example, Package Shipment contains multiple kinds of pickup-related requests and responses, which all are defined inside a schema for Package Shipment Pickup. This architecture is represented in figure 1:

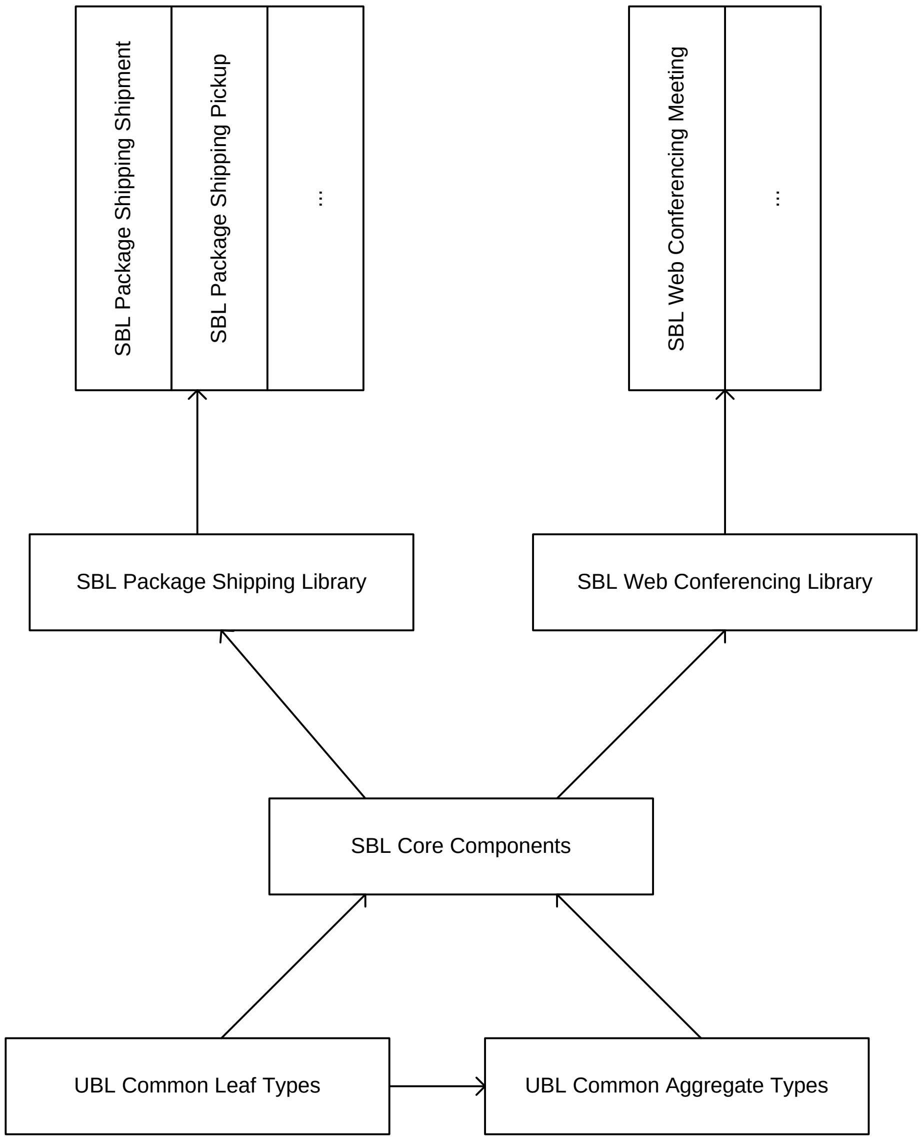
SBL Package Shipping Shipment
SBL Package Shipping Pickup
...
SBL Web Conferencing Meeting
...
SBL Package Shipping Library
SBL Web Conferencing Library
SBL Core Components
UBL Common Leaf Types
UBL Common Aggregate Types

Figure 1: The Architecture of the Services Business Language

Figure 1 illustrates the SBL architecture and dependencies between various schemas. In actuality, SBL encompasses more than the two service verticals that are illustrated in this figure. At the bottom are the UBL schemas for the most general and reusable types of information, which the SBL Core Components build upon. The SBL Core Components contains components that are reused across multiple verticals. Finally, each vertical contains a library of components that are reused in multiple schemas within that vertical, and one or more schemas for the various types of high-level documents (such as pickup-related documents) that are exchanged in transactions for that particular vertical.

Organizing a complex XML vocabulary like SBL into a set of smaller sub-vocabularies using namespaces makes them easier to maintain, and it allows an application to use only the parts of SBL that are required for a given transaction, resulting in faster load and validation times and a smaller memory footprint. For example, within the package shipment vertical, a tracking request – that is, a request to track a package that has been shipped – has no need for the definitions involved in scheduling a pickup. Breaking up the vertical group into multiple schemas makes this level of granularity possible, and so we broke the package shipment vertical, for example, into Shipment, Pickup, Tracking, Shipment Quote, Label, and a library of package shipment components that are reused across more than one of the other package shipment schemas. This builds upon the SBL library of components that are reused across verticals, and ultimately upon the UBL common leaf and aggregate types. Additional benefits of this layered architecture are that the library of components within each vertical group promotes reuse within that vertical, and the core library of SBL components encourages reuse of components across multiple verticals.

Requirements Analysis for SBL

After determining the domain space and the methodology to be followed in constructing SBL, we proceeded to the requirements analysis phase of Document Engineering. Ideally, a designer of an XML vocabulary should work directly with the providers of the services whose interfaces are being defined. We were only partly able to do this, and mostly with a Talaris product manager as an intermediary. As a result, most of our requirements work was "document driven" in the sense that our primary source of information for gathering requirements was the existing documents and forms that have evolved in offline business transactions. This vast body of forms and documents captures the required information for the business process of a service provider, and are themselves the result of requirements analysis informed by domain expertise. In this way, the documents serve as proxies for the domain expertise that is the optimal source of requirements information.

Simply looking for the presence or absence of a data element does not meet a high enough standard of document analysis. The obvious method of merely aggregating the data elements across separate definitions of requirements in a "document x data element" matrix will not result in useful logical models because it fails to capture the ways in which different data elements are used in different contexts. For example, it is essential to distinguish data elements that are mandatory in a context from those that appear only occasionally, and to distinguish data needed for operational purposes from that with more analytic or strategic value. This more nuanced work enables an analyst to aggregate data elements in ways that facilitate their reuse in different contexts.

Other sources of requirements that we consulted in creating our logical models were the web user interface (UI) that the providers currently provide for web-based transactions, and the APIs that they provide for direct transactions. There sometimes were discrepancies between the functionalities supported through the web UI and through the API. For example, in the web conferencing vertical, Web Ex, one of the service providers, allowed a user to set a recurring web conference meeting when using the web UI, but did not allow a recurring meeting through their XML API. In these cases, when functionality was provided through one medium but not another, we provided the given functionality as an optional part of the model, assuming that the application or a higher level transform would ensure that the document request was in the proper format for a particular provider over a particular medium. There were other cases, though, where an enumeration of possible values for a field differed between the web UI and the API. In these cases, we generally followed the API, since the API enumerations tended to be more recently developed than the web UI enumerations. Many providers, in fact, are just beginning to release XML APIs for the first time.

Designing the Logical Document Model

At some point in the modeling process, the modeler must decide what are the document-level components in the model. Document-level components are the root level components (also called a root level node) of complete XML documents. An XML document may have only one document-level component. This component, in essence, defines the entire document. In our case, an example document-level component was *ShipmentRequest*. An XML document with *ShipmentRequest* as the root node contains all the information needed to schedule a package shipment between the opening and closing *ShipmentRequest* tags. Such documents are referred to as “Shipment Request documents.”

We have already examined our process of dividing up the sub-areas within a service vertical into separate schema files for sub-vocabularies. This decision was made to avoid cluttering one schema file with relatively unrelated document level components, such as *PickupRequest* and *TrackingRequest*. This practice also avoids clutter of sub-components in top-level schema files. However, another consideration must be made regarding how many top-level components any document schema should contain. We made the decision that document schemas should contain the document-level components necessary to execute all the functionality in the area designated by the schema. In this regard, we considered it more important for each element to be self-contained – at the cost of some duplication of content between the request and the response – than to absolutely minimize redundancy between documents.

Another design question now arises from this discussion of document-level components. What should a document-level component contain? “All the information necessary to complete the transaction it describes” is only the obvious half of the answer. A second, critical question we faced has to do with the *amount* of content each root-level component should contain, or the real-world limitations on the use of that root-level component. Should a *ShipmentRequest* document schedule just one shipment? Multiple shipments? Would a shipment scheduled in a document containing multiple shipments be considered an independent shipment, or somehow related to its siblings? How does a response document deal with a request for multiple shipments? What if one of the shipments fails and others succeed? What if the shipments are to different recipients, or should be paid for by separate parties? All these questions have profound implications on how the components inside the *ShipmentRequest* document are designed. Components like *SenderParty* might have to move deeper inside the document so that the document can contain multiple different instances. A response, for example, may have to contain multiple container elements holding shipping labels and tracking numbers, in order to be able to match them up with the appropriate request elements.

Given the lack of clear answers here, we might argue eternally over the correct approach. The solution we chose is to model the electronic documents in such a way that they would be used in the same contexts or business processes as their physical counterparts – that is, we modeled the electronic documents after the real world event that the paper document represents. The paper form from Federal Express used by a mailroom employee to schedule a shipment only contains a space for one sender and one recipient, and a shipment via FedEx or DHL, for example, goes from one party to one party. A modify shipment form on the web will let a user modify only one shipment at a time. Thus, we chose to follow the restrictions imposed by the business event that is encoded by the real world documents when designing our own document elements. We think that this approach, in addition to helping designers model documents, will improve adoption of the documents by the implementers of the web services that use them.

Designing Logical Components

Components are meant to be re-usable data structures. They could be defined in a document schema, but they are most often found in a library schema file. This allows them to be re-used in multiple document schemas. For example, our package shipment library contained a component called Shipment. Elements of Shipment type could be used in any of the package shipping vertical's document schemas

What makes a component a component? We decided that a component is a flexible, semantically justified modeling of a real world object or service. A good illustration of the process we used to make this kind of decision comes from our experience modeling a Package. There are many ways to model a Package. It can be a type of container with dimensions and weight, or it might include the above three pieces of data along with sender and receiver addresses. It might even include a shipping service type and a tracking number. At some point, the Package component will become too large if it keeps being expanded in this way – you probably don't want it to include billing information, but where should the modeler draw the line? Here are a few strategies we found helpful:

1. Follow Provider Requirements: One good way to determine what a component contains is to look at the provider's information requirements. For example, we found that all three of the shipping providers we studied required that all packages in a multi-piece shipment share the same origin and destination. This enabled us to pull addressing information up out of the Package component and put it at the child level of Shipment.

2. Be Flexible Across Providers: Next, a component needs to be flexible across providers. This means that our definition of what a Shipment is must fit the definitions for a shipment of all our providers. One provider might think of a shipment as one package going from one place to another with one set of services. Another might allow for multiple packages with one origin, one destination, and a set of services (and, once scheduled, they might share the same tracking number or each have their own). Still another might let you send multiple packages to different places in a single shipment. We took an iterative approach to this problem. We started by modeling a Shipment that fit one provider's definition, and gradually modified it into a more general model. This gave us a model locally optimized for our set of providers.

3. Separate Essential and Variable Information: Returning to Package, we want to decide whether or not the shipping service – something like Fedex Overnight or DHL Express – is part of the *Package* component. There are arguments for either approach. One commonly hears locutions such as, "I got an Overnight FedEx package today!" But at the same time, one also hears, often from the same person, "I need to send this package", meaning the shoebox-sized box that weighs 3 pounds. Which is the correct use of the word package? We found that it helped to separate the package's essential characteristics, like packaging and weight, from its variable characteristics, such as delivery type, signature service, and so on. We then created another component, ShipmentServiceType, that contains the variable information. This component is aggregated with a Package component(s) within a Shipment component.

4. Look for Utility and Re-Use: Components that model things like a package and complex services like a shipment have dominated our discussion so far. Not all components are of this kind, though. Talaris intended that the Services Business Language would contain a library of components useful across many service verticals. For example, a customer scheduling a courier pickup will submit a time range and address to the provider, specifying that the courier should arrive at a specific place during a specific time period. We modeled a component called *TimeAddressSet* that combines a UBL *Period* component with a UBL *Address*. We soon realized that this component would be useful for modeling the limousine and car services in the SBL Travel vertical, and placed it into our Core Component Library.

To Reuse, or Not to Reuse?

A modeler using UBL or other existing XML vocabulary may encounter situations in which he or she must choose whether to use an existing component to describe some part of the model. We made extensive use of UBL components in our work: *AddressType*, *ContactType*, and *CodeType*, for example. However, we chose to create our own Shipment type rather than use UBL's. Here are some strategies we found useful for our efforts:

1. Does the existing component match your requirements? This is the most important question to ask when unsure about using a library component. There is no reason to use a component whose data model contains extraneous information just because it has the same name as a component you wish to define; the name can be a distraction that obscures differences in the logical models. We chose to implement our own *Shipment* component in lieu of the UBL's shipment because the UBL component could not describe information we needed to describe.

2. Will you be using the component correctly? A second consideration is whether the intended use of the UBL library component matches how you will use the component in your model. Even if a library component contains the information your model needs, it should not be used in ways that don't conform to its intended uses. To do so would be "tag abuse" that reduces semantic precision and thus should be avoided.

3. How complete is the library component? When a library component does not seem to be complete enough to describe the data you need to convey, you can either extend it or aggregate it into a larger component. One example arose when we searched in UBL for a component we could use to describe payment information for a shipping transaction. The existing UBL type, *PaymentMeans*, seemed to be designed specifically to describe actual funds transfer from one party to another. There did not seem to be a way to make it fit the preferred method of payment in the shipping vertical: billing a preexisting credit account with the provider. Paying in this manner involves no immediate funds transfer. Therefore, we created a new aggregate component that contained a UBL *PaymentMeans* element and another element we created called *CreditAccount*. This new component is useful in many contexts, so we included it in the SBL Core Component library.

4. Can the component be restricted or extended? We made some use of XSD extension and restriction in our schemas, both of our types and UBL types. UBL types are designed to be easy to extend and restrict [GREG 2002]. The *Address* type, in particular, lends itself to restriction, since every single child element it contains is optional. We restricted *Address* to create a *SimpleAddress* type. We did so because in some situations the only address information that should be communicated is a city, country, and perhaps state and post code. When tracking a package, the locations of the "hops" the package takes between origin and destination only need to be listed in the following form: "Oakland, CA, USA". Furthermore, the *SimpleAddress* requires that there is a city and a country, unlike the optional cardinality of these elements in *Address*. Restricting UBL's *Address* in these ways makes our model semantically tighter. .

Implementing SBL's Logical Models as XML schemas

After creating the conceptual models of the SBL domain, the next step was to implement the model in the concrete syntax of an XML schema language. This means taking the abstract (implementation neutral) model of the domain, represented with UML class diagrams or some other modeling notation, and translating that model into an XML schema language. At this stage in the (im-)maturity of the art of designing XML vocabularies, there is too little agreement in how to complete this critical task. This section will briefly outline some of the design issues that we faced in moving from an abstract conceptual model of the domain to an actual implementation.

1. Choice of XML schema language. Like UBL, the language we chose to use for implementation was the W3C XML Schema. This allowed us to directly leverage and build upon UBL's XML schemas. It seemed obvious to consider the XML schemas published by the UBL team as the "UBL Library" but only afterwards did we learn that this view was only partly correct. What UBL considers its "library" -- the reusable general business information entities (BIEs) – is defined in a syntax-neutral manner in a spreadsheet form. The model expressed by this spreadsheet is then transformed into an XML Schema (XSD) file that defines the BIEs as aggregate types useful for a specific document context. Similarly, Carlson [CARL 2001] advocates the automated creation of XML schemas from a modeling notation (in his case, from UML). But we believe that XML schemas carefully crafted "by hand" are of higher quality than those we might have produced by programmatic transformation.

2. Use of XSD features. In the process of translating from the conceptual model to a particular implementation, there were many choices to be made about which aspects of the XML Schema Language to use and how best to represent in XSD a particular model. For example, XSD has multiple mechanisms for dealing with variable content, among which the chief methods are substitution groups and type substitution with the xsi:type attribute. We tried to follow UBL's recommendations about how best to use XSD to encode models, and we believe that these will undoubtedly become standard practice [MALE 2002].

3. How to enable extensibility. Another issue that presented itself in implementing the models in XSD was the issue of which XSD structures should be used in order to make components extensible and reusable.

4. How to express application business rules. Other implementation issues are concerned with higher-level application functionalities. For example, XSD is not capable of expressing many kinds of constraints that a schema author might like to express (e.g., co-occurrence constraints). One might choose to supplement XSD with another schema language, such as Schematron [SCH 2003], or one might accept that the application will have to do certain kinds of validation itself

5. Names for types, elements and attributes. Other issues that presented themselves in the process of implementing the models in XSD were matters of style. One such example is how to handle capitalization of multi-word elements and attributes. Like UBL, we chose to use lowerCamelCase for attributes, and CamelCase for everything else – that is, elements and types. Another stylistic issue that we had to contend with was the question of whether and when to qualify element names with the context – for instance, whether a Sender element inside a Shipment element should be named *Sender* or *ShipmentSender*. In this regard, we followed UBL again, and omitted the Object Class *Shipment*, using only the Property Term *Sender*, when it was clear what the Object Class was. In the example given, it is clear that *Sender* is the Property Term of the Object Class *Shipment*, and so we omitted Shipment and called the element Sender.

Reflections on Using UBL

Overall our experience using UBL was positive. The UBL work that we were able to leverage in working on SBL resulted in better models and much more rapid development time. The common leaf types are very comprehensive and versatile. The library is a bit more complicated to use, but much of our difficulty was due to our over-reliance on the published schemas instead of the logical model in spreadsheet form. As UBL matures and is "packaged" with more design advice, tools, reference implementations, and so on, we predict it will be an invaluable resource for developers of XML vocabularies.

Summary

Web services depend on XML documents that can be understood by the parties (or applications) that wish to do business, which requires that they agree on the meaning and organization of the information elements the documents contain. Without an agreement on an XML vocabulary, the glib promises of "easy integration" and "interoperability" with web services are undermined by the need for custom coding to transform instances of one side's data model into the other's. The development of an

XML vocabulary of practical scale is a complex undertaking that is even more challenging when reuse of another XML vocabulary is a design goal.

In this paper we have discussed the design of the Talaris Services Business Language™ (SBL), an XML vocabulary developed to make it easier for suppliers of business services to integrate with the Talaris application, which is used by corporate employees to purchase services on the Web. Talaris chose to use the emerging Universal Business Language (UBL), a "horizontal" XML vocabulary of common semantic components, as a basis for SBL to give it more robust semantics and to encourage its adoption.

Because of our focus on the "Document Engineering" methodology of vocabulary design, we considered specific features of SBL or UBL only to illustrate the methodological and design issues for developers facing similar challenges.

Acknowledgments

The first two authors worked for Talaris Corporation when this work was carried out; the third author has been a technical advisor to Talaris since its founding in 2000. We thank Adam Blum (VP, Engineering) and Patrick Grady (CEO) of Talaris for encouraging us to write this case study paper.

We also thank Mike Adcock, Arofan Gregory, Eve Maler, Tim McGrath, and other members of the OASIS UBL initiative for helpful comments on our work and on earlier versions of this paper.

References

[CARL 2001]. Carlson, D. *Modeling XML Applications with UML. Practical e-Business Applications*. Addison-Wesley, 2001.

[GLUS 2002]. Glushko, R., and McGrath, T. Document Engineering for e-Business. *ACM Symposium on Document Engineering*, 2002, 42-48.

[GREG 2002]. Gregory, A., & Gutentag, E. XSD Type Derivation and the UBL Context Mechanism. *IDEAlliance XML 2002 Conference*, December 2002. http://www.idealliance.org/papers/xml02/dx_xml02/papers/05-05-06/05-05-06.html

[HDI 2003]. Harley-Davidson. How to Become a Harley-Davidson EDI Trading Partner. https://www.h-dsn.com/foundation/content/pdf/howto.pdf (as of 1/4/2003).

[IEEE 2002]. *IEEE Computer*. Call for papers: Special Issue on "Web Services Computing," http://tab.computer.org/tfec/webservices (as of 1/4/2003).

[MALE 2002] . Maler, E. Schema Design Rules for UBL...and Maybe for You. *IDEAlliance XML 2002 Conference*, December 2002. http://www.idealliance.org/papers/xml02/dx_xml02/papers/05-01-02/05-01-02.html (as of 1/4/2003).

[SCH 2003]. Schematron: An XML Structure Validation Language using Patterns in Trees. http://www.ascc.net/xml/resource/schematron/schematron.html (as of 1/4/2003).

[TAL 2003]. Talaris Corporation. Solution Overview. http://www.talaris.com/html/solution/overview.html (as of 1/4/2003).

[UBL 2003]. UBL: The Next Step for Global E-Commerce. http://oasis-open.org/committees/ubl/msc/200204/ubl.pdf (as of 1/4/2003).

[VDV 2002]. Van der Vlist, E. *XML Schema*. O'Reilly, 2002.

[XCBL 2003]. XML Common Business Library. http://www.xcbl.org/about.html (as of 1/4/2003).

[XMLO 2003]. XML.ORG Registry. http://www.xml.org/xml/registry.jsp (as of 1/4/2003).