The Talaris Services Business Language™: A Case Study on Developing XML Vocabularies Using the Universal Business Language

Calvin Smith
Patrick Garvey
Robert Glushko

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

27 September 2002 (revision 4)

Abstract: The development of an XML vocabulary is a complex undertaking. This paper outlines the methodology used in the development of the Talaris Services Business Language™ (SBL), an XML vocabulary for web-based services procurement developed using the reusable semantic components of the Universal Business Language (UBL). UBL is still an incomplete standard, and the complete UBL model is not yet available in a readily accessible format; many of the decisions we made reflected this reality. Lastly, we discuss the design and implementation challenges we encountered and propose rules and guidelines for similar projects.

I. Definition of the Problem

Services Business Language

The Talaris Services Business Language™ (SBL) is a library of XML document schemas designed to enable service procurement transactions. Companies engaged in business-to-business transactions exchange information to complete a transaction, such as the sale of a good, often in the form of a request document and a reply document. This information is typically entered through a web form, put into a batch file, or communicated by invoking an API. An increasingly popular form of communication and online business is through XML document exchange.

Online business so far has consisted primarily of application-consumer transactions using HTML. The next generation of e-commerce applications is increasingly designed for application-to-application transactions, and XML is rapidly becoming the lingua franca of this new breed of e-commerce applications. Defining data formats in an ad hoc, proprietary, manner is not sufficient for inter-application communication between a wide-range of ever-changing companies, and XML, with its formality and flexibility to describe arbitrary data formats, fills this niche.

An XML document can be constrained by associating it with an XML schema. A schema might state that a document must contain one <Person> element, and that a <Person>

must contain a <Name> and <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. A schema may also enforce datatype restrictions, such as specifying that an <Age> element must contain integer rather than floating-point values.

XML's flexibility to describe documents of arbitrary format is its greatest strength, but it is also its greatest weakness. Since XML has no fixed semantics, it can be used to describe anything. But this means that there is no standard way to describe anything, and different vocabularies may define the same objects with different elements. One person's <Price> might be another's <RetailPrice>, and so on. The difficulty is compounded when complex entities like <Shipment> or <Flight> need to be described.

Electronic commerce on the Web is greatly facilitated by common document designs. If such common documents are created and agreed upon by parties wishing to do business, communication becomes much easier. Even better than individual trading partners agreeing on document designs are efforts to create industry standard XML vocabularies for e-business. These standards might be specific to a vertical industry, such as the Open Travel Alliance's efforts in the travel industry, or HL7's work in the healthcare industry. Or they might, like the XML Common Business Library (xCBL) or the Universal Business Language (UBL) (more on this later), attempt to define information components that are common to documents in any business domain.

A need, perceived by Talaris, for a standardized library of components designed for the electronic procurement of services motivated the development of SBL. In order to remain aligned with the emerging UBL standard, we chose to build SBL using UBL types. Initial SBL design work focused on package shipment and web conferencing service verticals, both of which are offered through the World Wide Web by providers such as FedEx and WebEx. In addition to components specific to these verticals, we constructed core components that could be used for services procurement in other verticals.

Use of UBL

The Universal Business Language, or UBL, is a set of XML document components being created by the Organization for the Advancement of Structured Information Standards (OASIS) standards body for use in electronic commerce. The following is the UBL Committee's Statement of Purpose:

> The purpose of the UBL TC is to develop a standard library of XML business documents (purchase orders, invoices, etc.) by modifying an already existing library of XML schemas to incorporate the best features of other existing XML business libraries. The TC will then design a mechanism for the generation of context-specific business schemas through the application of transformation rules to the common UBL source library. UBL is intended to become an international standard for electronic commerce freely available to everyone without licensing or other fees.

Talaris chose to use UBL as the basis for SBL. Doing so was advantageous for three reasons:

- Talaris can use UBL's definitions for common business components, rather than defining identical components
- By implementing with UBL Talaris aligns itself closely to an emerging 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.

Currently, UBL uses a three-layered architecture. At the top level, there is a schema and namespace that defines what UBL, in the *Modularity, Namespaces and Versioning* paper, 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.

Our view of the UBL Library was informed by the XSD schema files made available on the UBL website. We learned later that this view is an incorrect one to take. There is a general UBL model available only in spreadsheet form that defines the general business information entities (BIEs) used by UBL. 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. The current XSD version of the library expresses these BIEs in the context of an Order document. However, we assumed that the XSD file was the definitive UBL model for aggregate types. Many of the difficulties we encountered using the Library stem from our overlooking this application of context. In reflection, we believe that a more accessible general model would be a great aid to designers who are trying to adopt UBL.

The goods procurement focus of the current version of the UBL Order Library made it inappropriate for us to reuse its high-level components, such as <OrderRequest>. However, many of the aggregate components in the Order Library and the entire contents of the common leaf types schema provided an excellent foundation upon which to create SBL. In addition to using UBL components, we also tried to follow recommended UBL methodologies for how we divided our schemas into semantic groupings called namespaces, and how we implemented our schemas in W3C XML Schema. One recurring difficulty, however, is that the UBL standard is still under development and subject to change.

After deciding on using UBL wherever it is appropriate, there is still the question of when a UBL component is a close enough match to the needs of a particular context, and when there is enough difference that a custom-made component is preferable to reusing the UBL component. This issue will be addressed in section IV, "What is a Good Logical Document Model?"

II. SBL Structure

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. 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 domain into sub-groupings. 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. We have created a library of SBL Core Components that can be used across verticals.

The next decision was whether to break the vertical up into smaller pieces. UBL, in the *Modularity, Namespaces and Versioning* position paper, recommends that the high-level document schemas – what UBL calls a functional area – be fairly modular and contain no more than ten or so types. Accordingly, 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:

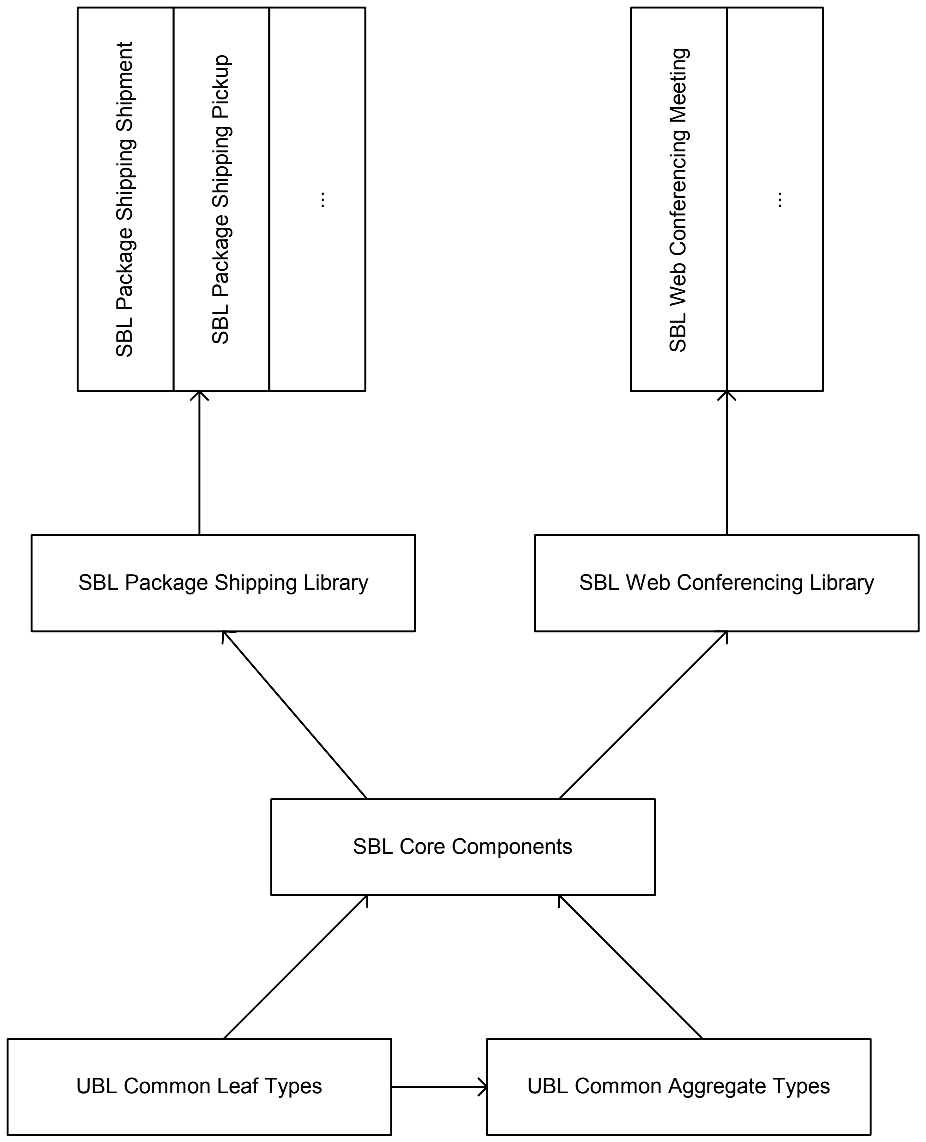
SBL Package Shipping Shipment
SBL Package Shipping Pickup
…
SBL Web Conferencing Meeting
…
SBL Package Shipping Library
SBL Web Conferencing Library
SBL Core Components
UBL Common Leaf Types
UBL Common Aggregate Types

Figure 1: SBL Architecture

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, 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.

The advantage of having this level of modularity is that it breaks the schemas up into more manageable pieces that are 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 – should not have to load the type definitions for 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.

III. Requirements Analysis

After determining the domain space and the methodology to be followed in constructing SBL, we proceeded to the requirements analysis phase. This phase of our work process was document driven and structure independent. 'Document driven' means 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 often capture 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 standard source of requirements information.

The 'structure independent' factor means that instead of copying any one provider's document or form structure for a given interaction, we constructed harmonized documents that captured the data models expressed by all the providers' documents but weren't optimized for any one providers' model for that interaction. In deciding how complete to make our models, we followed the 80/20 rule, and focused on the 20% of the model that covers 80% of use.

Other source 'documents' that we consulted in creating our 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.

IV. What is a Good Logical Document Model?

Document Level Elements

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. 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. More simply, this means that a schema file called "tracking.xsd" should contain root-level elements called <PackageTrackingRequest> and <PackageTrackingResponse>, since these are the two interactions related to tracking. Each of these two root level elements contains all the information required for that particular part of a package tracking transaction. 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 more complicated schema for Shipment-related activities contained a root-level component called <ShipmentCancelRequest>, even though this element was of a generic CancellationRequestType that was defined in another schema. We chose to define a <ShipmentCancelRequest> element in our document schema because cancellation is a shipment-related activity. Again, we considered it more important to have the Shipment schema be self-contained than to absolutely minimize redundancy. The CancellationRequestType is defined in another schema (SBL Core Components) so that it can be reused for different kinds of cancellation requests. The documents traveling between trading partners are also semantically richer if the name of the root node of a shipment cancellation request is <ShipmentCancelRequest> as opposed to <CancelRequest>.

Not surprisingly, the answer to the next question about document-level elements follows from the previous example. What are the document level elements, and what can they do? Document-level elements in each document schema should be for documents describing each of the possible activities in the functional area defined by the schema. If it is possible to schedule, change, and cancel a conference call, and each of these transactions involves a request and reply document, the schema devoted to conference calls should contain <ConferenceCallScheduleRequest>, <ConferenceCallScheduleResponse>, <ConferenceCallChangeRequest>, <ConferenceCallChangeRequest>, and so forth.

A third 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 will have to contain multiple elements holding shipping labels and tracking numbers, and be able to match them up with the appropriate request elements.

These are all solvable problems, and one can argue eternally over the correct approach. However, there is an easy solution to the dilemma: model the electronic documents to be used like their physical counterparts and after the real world event that the 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 even that is encoded by the real world documents when designing our own document elements.

Header and Details Division

Breaking the document into a header section and a details section follows the example of xCBL and UBL, both of which separate the data about the actors in a transaction from the data about the transaction. In SBL, we divided the documents into a header and a details element, both of which are container elements for their respective information. The header element contains the actors information – at minimum the requesting and provider parties and the date of the request. The details section contains the substantive body of the transaction, i.e., the content of the transaction – for example, the sender and recipient of a package, and the type of service for the package (next day AM, and so forth).

Within the details section of each document, we followed the general principle of grouping together information that belongs together, using container elements, and adopting hierarchical rather than flat structures wherever possible. The decision about what to group together basically came down to grouping like things with like things. For example, all the subelements of an address belong in a container element called *Address*, and at a higher level, all the elements that comprise a shipment belong in a logical grouping, *Shipment*. This results in more logical groupings of components, wherein the structure of the information is made evident by the hierarchical structure of the elements, and easier access to those components through XPath expressions.

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.

Shipment is a component we defined for the SBL Package Shipment component library. The focus of our discussion here will be on two issues: 1) what makes a component; and 2) when components should be designed from scratch and when existing components in a repository such as UBL should be reused.

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 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 Package and complex services like Shipment have dominated our discussion so far. Not all components are of this kind, though. We 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.

A modeler using UBL may encounter situations in which he or she must choose whether to use a UBL or custom 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 library 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. 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. Not only did it lack data elements necessary for our model, but in accordance with goods procurement focus of UBL, it seems designed to model a freight shipment, and not a package shipment. As noted before, though, the version of the UBL library we used expressed UBL types in the context of an Order. Expressing the UBL model of a shipment in the context of Package Shipment would have been a better use of the UBL library.

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. Incorrect use of a library component should be avoided always, even if the component contains the information your model needs. The practice of using a data component for something other than its intended purpose is known as "tag abuse," and should be avoided. For example, in our first design of the Shipment component, we defined the Sender and Recipient elements inside Shipment as being of UBL Party type. A UBL Party contains a name and an address, and can be used to describe an entity. However, we later learned that this was an incorrect use of the type, its intended use being to designate the actors in a business transaction. According to the UBL methodology, Parties belong at the top header level of a document, not inside content components. All we really needed inside the Shipment component was a name and an address, so we created an aggregate component out of two UBL components – Contact and Address – and called it <MailingAddress>. In retrospect, we correctly applied context to UBL aggregate BIEs to make our own aggregate BIE, though we weren't aware of doing so at the time.

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. This is similar to what we did with the aforementioned <MailingAddress> component. Another situation arose when we searched 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. At the end of the month there may be a transaction akin to the type described in the UBL component, but not during each transaction. 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. 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". Restricting UBL's address makes our model semantically tighter. Elsewhere, we extended and restricted our own components. For example, all our documents share a single Header type that contains a Requesting and Provider party along with a date. However, the header in Shipment request documents should also contain payment information, and perhaps a RecipientParty and SenderParty. Instead of re-defining a completely new header, we created a new ShipmentRequestHeader type in the Shipment namespace that extended our core header by adding the necessary elements.

Finally, we should comment on the emphases that appear in the UBL schemas and the effects these emphases had on our efforts to use UBL content in our models. UBL as it exists now in XML schema form is designed, or optimized, for a goods procurement context. We have already discussed how the base Party type contains elements that are specific to ordering and delivering goods. This goods procurement emphasis did not have too great an effect on our using the components in the shipment vertical, since this is a similar area dealing with many of the same things as direct goods procurement: Addresses, Contacts, Dimensions, Quantities, and so forth. When we applied UBL to the Web Conferencing vertical, we found that the UBL components were less useful. Web conferencing is a vertical whose entities mainly contain non-tangible content – 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 like we saw in the Package component. Most of the parts of a Web Conferencing Meeting component seem like options, strings, or numbers. The Web Conferencing domain is theoretically infinite in scope as providers can add new features to their software at will. This problem will probably recur in other event-based verticals. Using the UBL common aggregate types (UBL Library) was less useful in this situation, but the core leaf types (Core Components) were still quite useful.

This infinite extensibility illustrates one final problem we faced: how to describe endless options. Services procurement documents are especially prone to endless options, as they often contain fewer descriptions of physical objects than do goods procurement documents. When everything about a service seems to be an option, there are two extreme methods the modeler may use. One is to give every option an element naming it – <RecordingOption>, <ApplicationSharingOption>, and so forth. The other is to call everything an <Option> element and vary a "name" attribute or the content of the element. The first method makes the model less extensible, since every time you want to add a new option you must create a new type. Not to mention that such a document would be terribly complicated to transform if there were more than a trivial number of options! The second method is too generic – if every element is called <Option> we get no meaning from the names of the elements themselves. This makes the model much less descriptive of the data it contains. Of course, there is a balanced middle ground that we recommend. We chose to group the options into containers according to their similarity: <FeatureOptions>, <RecordingOptions>, etc, and have each container hold single options whose content indicated the precise name of the option. We felt that this was a good balance between remaining generic but still preserving meaning in the names of our elements.

V. Implementation

After creating the conceptual models of the domain, the next step was to implement the model in some particular XML schema language. This means taking the abstract (implementation neutral) model of the domain, represented with UML class diagrams, and translating that model into an XML schema language. This section will outline some of the design decisions that we faced in moving from an abstract conceptual model of the domain to an actual implementation.

Like UBL, the language we chose to use for implementation was the W3C XML Schema. This allowed us to directly leverage and build upon the UBL schemas. Furthermore, W3C XML Schema has the imprimatur of the W3C, and is the closest thing to an XML schema standard, other than DTDs, that there is at present. One last consideration is that W3C Schema is currently the schema language (other than DTDs) that is most supported by XML application vendors and software developers.

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. The advantage of substitution groups is that the instance documents are clearer, since instead of <Document

xsi:type="Label">…</Document>, we can just specify that Label is part of the Document substitution group (and derive <Label> from <Document>) and use <Label>…</Label>. Type derivation, on the other hand, is not so clear in the instance document, but is a more robust alternative than substitution groups, since the relationship between the two elements is completely explicit. As type-aware XML tools become available and type-aware programming becomes possible, there will many other advantages to using type substitution over substitution groups. The UBL position papers do not specify a position on variable content mechanisms, but a UBL committee member commented to us during a design review that UBL was intentionally avoiding substitution groups in favor of type substitution, so we chose to follow UBL's example.

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. For example, in cases where there is a choice between several elements, xsd:choice seems like the perfect solution, but it can also be argued that xsd:choice should be avoided whenever possible, since a choice group cannot be extended so that more elements are available in the choice. This severely restricts the extensibility of any type or element that uses xsd:choice. Using xsd:sequence, in combination with minOccurs="0" for all child elements, it is possible to extend the sequence of elements and thus extend the choice that the sequence represents, and so we adopted the sequence method of having extensible choices.

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, or one might accept that the application will have to do certain kinds of validation itself. In the example given above, where we use the pattern of an xsd:sequence with each element's minOccurs set to '0' in order to emulate an xsd:choice, an XML instance fragment that had no elements would be valid, even though in a given instance document, at least some of the content should be required. In UBL, this occurs in the AddressType, for example, which has twenty sub-elements, all of which are optional.

In such cases, there are four choices about how to deal with all-optional content. First, we could just not validate. Secondly, we could consider that the application will ensure that elements that should be present are present. Thirdly, we could use the xsd:key mechanism for certain kinds of occurrence constraints. And fourthly, we could supplement XML Schema Language with a rule-based schema language, such as Schematron.

The first alternative, to not validate, is a viable alternative only when you are certain that the XML content is well-formed and validation is not necessary. In most cases, some kind of validation is desirable, so the first alternative is not a general solution. The second alternative, leaving it to the application to validate than an <Address> has children elements, is a possibility, but it is always preferable to validate as close to the source as possible, and only let the application deal with valid documents. The third option, using xsd:key, could only cover some of the situations in which occurrence constraints were needed, and since the xsd:key mechanism is intended to be used for key constraints primarily, using it for occurrence constraints seems like "tag abuse." This leaves the use of Schematron as our remaining alternative.

Schematron is a simple rule-based XML schema language that is especially good for validation tasks that XSD is ill-suited for. As an example of how Schematron could be used, consider an <Address> element, which is of the UBL type, AddressType. It has a content model of twenty or so optional elements. We would like to ensure that an <Address> element always has some content, but this is not easily expressible in XSD, even if we tried re-writing the AddressType from scratch. Using Schematron, though, we can create a simple rule that states that an <Address> element must always have at least one child element, of any kind. If it doesn't, it will fail validation under Schematron. This, in fact, was the approach that we chose when XSD was unable to express certain validation constraints and we weren't certain of the validity of XML documents we were using.

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.

One last matter of style that we had to decide on was how to handle code lists. The UBL position paper, *Code Lists*, served as a model, and we adopted the scheme outlined there. This scheme is to include the source of the code list in the element name, and to include that element in a container element. The example given in the paper is the following:

```
<LocaleCode>
        <ISO3166Code>code</ISO3166Code>
</LocaleCode>
```

In SBL, an example of a code list that we derived from an outside source is the Hazardous Delivery Type code list. This code type appears in an instance document in the following manner:

```
<HazardousDeliveryCode>
        <US49CFR172.101Code>code</US49CFR172.101Code>
</HazardousDeliveryCode>
```

Another question about the use of code lists was whether the codes in a code list have to be abbreviations for a more complete value, like '1' stands for 'Next Day AM' service, or whether the code could actually be what it represents, 'NextDayAM' for 'Next Day AM' service. The Code List position paper did not address this point specifically, but a UBL committee member stated that the second use above is definitely within the scope of the intended use of code lists in UBL.

Some of the code lists that we used in SBL were from outside sources, such as the Hazardous delivery code list given above, which came from a standard Department of Transport code list for hazardous deliveries, but others were provider specific or aggregations of provider code lists. For example, each provider has its own code list for identifying the delivery type of a package – e.g., 01 for Next Day Air (UPS), and X for Express (DHL). In these cases, we used the human-readable format of what the code stands for, and codified each provider's code list individually. For Delivery Type, this means that it has the following child elements, all of which are optional: UPSDeliveryType, FedExDeliveryType, and DHLDeliveryType. Each of these enumerates the code lists for that particular provider in a human-readable format. This solution makes extensibility quite simple, as another provider can easily be added via xsd:extension.

VI. 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 order context instead of the logical model in spreadsheet form. Some thoughts:

1. The only current XML Schema implementation of UBL is context-specific for an Order document. We were unaware of this fact, and our mistaken interpretation of context specific components as universal influenced how we used the library. If anything, this demonstrates a need for a more accessible representation of the context-neutral library (now contained in an Excel spreadsheet) that designers can use to build their own documents.
2. The UBL XML schemas that were available to us in our development efforts were optimized for procurement of goods. PartyType, for example, has shipping and ordering and tax information elements in it. This makes in unsuitable as a basis for other types of parties that aren't involved in goods procurement. For instance, a party booking a restaurant information has no need for shipping, ordering, or tax information, and a party involved in a non-commercial transaction has even less need of much of the UBL party. In this case, a more primitive type such as Entity might be desirable, and non-commercial parties, such as a Student, could then build on Entity. Both Party and Student could extend Entity. We had this problem when we wanted to find a good way to describe actors in a transaction that aren't quite Parties who engage in the transaction but are merely passive recipients of some of the transaction's results. UBL might be more universally applicable if it used extension of smaller types rather than the "kitchen sink" approach they use now.
3. As related to #1 above, the components in the common aggregate types schema (UBL Library) worked well in the traditional vertical we took on: Package Shipment. We used Party, Address, PaymentMeans, etc. These components were not as useful in the Web Conferencing vertical, where there was no good way to wrap some more traditional business things around the many web conferencing options. This might be a recurring problem as we use UBL in more places, especially places that are far

removed from the context of goods procurement. The common leaf types, however, were equally useful in both verticals, and we anticipate that they will be equally useful in all domains, since they are the low-level primitives from which any domain builds its aggregate types.

4. There is no best practices document for using UBL components. There should be guidelines on how they should be extended, restricted, used, etc. Should xsi:type be used when declaring an instance of a derived type, or should substitution groups be used, etc. The UBL people seem to have answers for these questions, but they aren't codified or readily accessible by UBL's adopters. We relied on informal advice from UBL sub-committee members, but a best practices document would have saved a lot of effort.
5. A transform for the documentation in the UBL schemas would be very nice. Also, a high-level explanation of the intended use of each element or type would be helpful. We had some confusion with PaymentMeansType and CodeType, and it wasn't clear from the documentation how these components were supposed to be used.
6. Domain knowledge is critical, especially for the more complex components. We needed at one point to define what a Shipment should be, and went to the definitions each provider has, constructing something that could fit all three. Still, though, there were times when it was difficult to distinguish between one provider's idiosyncratic definition of a component, and a normative core definition that domain experts would tend to agree on. At such times, there is no substitute for having access to a domain expert.
7. The UBL website is not very user friendly, and is only updated infrequently. To check the status of the next release, for example, we had to view the subcommittee internal email list to see what people were saying in their emails.
8. We started by looking closely at just one vertical, and were asked to begin work on a whole library – a multi-year effort. This might have been a bit ambitious. In retrospect, it would have been better to look at all the verticals and then begin from the ground up. However, our task was to complete the schemas for the two verticals in a short time frame, and we probably would not have been able to accomplish this had we spent a good amount of time reviewing all the verticals. In short, it is always best to do requirements analysis of as much of the domain as possible: this results in more flexible components, greater reuse, better models, etc. Sometimes real-world time constraints prevent this kind of exhaustive analysis, and components have to be adapted as they are applied to a broader domain, which was the case with the development of SBL.

VII. 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. We thank Adam Blum (VP, Engineering) and Patrick Grady (CEO) of Talaris for encouraging us to write this case study paper.