Re-format page sizes

Developing Vocabularies with Multiple Distributed Teams

Jim **Gabriel**

Abstract

Sharing the development and maintenance of XML schemas and transformations across multiple developers presents a number of challenges. We think and code in terms of objects – customerID, address, etc. – yet capture all required combinations and permutations of these objects in clumsy schema and transformation files. As a deployment mechanism, files are acceptable. However, when development of those files is distributed over multiple teams and locations, the file as a container mechanism becomes unacceptable.

To support multiple distributed development teams, a development environment for schema families must support the concept of object-level management of the objects described by the schemas. This raises the need for a truly single-source object modeling environment, presented to the developer community in a multi-user working environment. Via a robust check-in and check-out mechanism, developers should work in private single-user workspaces, accessing multi-user resources in a central view of the model. This necessitates locking strategies, conflict resolution, impact analysis, coherent object-level version management, user groups, permissions, user roles, a release mechanism, and a deployment mechanism.

This paper describes the constraints governing multiple developer environments for schema families, explains each of the above requirements, and explores in detail the process and workflow for those involved. Where relevant, the presentation of this paper draws on case studies from actual projects, and provides recommendations for best practices in collaborative schema development projects.

RenderX
XSL • FO
formatter

Re-format page sizes

Table of Contents

1. Integrating Data Models in Schema Families 3
- 1.1. Externalized, Standardized, Federated 3
- 1.2. Active Metadata 4
- 1.3. Impact Analysis 4
- 1.4. Necessary Layers of Abstraction 4

2. Fragility Through De-normalizing 6

3. Model-Driven Metadata Management 6
- 3.1. Round-tripping Through a Non-XML Model 6
- 3.2. Specific Requirements of a Model to Support XML schema Management 7

4. Functional Requirements 7
- 4.1. Import 8
- 4.2. Editors 9
- 4.3. User Access Control Layer 10
- 4.4. Version control 11
- 4.5. Export 11

5. Automating Assembly 11

6. Summary 12

Bibliography 12

RenderX
XSL • FO
formatter

Re-format page sizes

1. Integrating Data Models in Schema Families

When multiple developers collaborate on the development of schemas, a complex management situation arises. The process of achieving the required results when sharing development presents challenges, both in terms of the way human beings need to interact and in terms of the technical infrastructure necessary to support that interaction. Furthermore, the resulting family of schemas rarely represents a new set of constraints, being more usually an interpretation of a set of existing business requirements in XML schema, in order to achieve in implementation-independent way of describing the rules of a given process or set of processes.

This paper takes as its opening example the collaborative development requirements that arise when developing applications in a service-oriented architecture (SOA). The reason for this is that successfully implementing a SOA is possible only when the conceptual requirements of the processes and data flows of an organization are charted and visible in an integrated enterprise data model. The integrated data model is an essential precursor to the SOA, and must integrate all existing underlying data models and business processes, and allow for considerable future expansion. The primary reason for creating an integrated data model before implementing the SOA is to insulate services from underlying technologies by enabling developers to code to implementation-independent contracts as opposed to application-specific interfaces.

Based on this premise, and the global, de facto adoption of XML schema standards for managing the technical implementation of Web Services (WSDL et al), the visible expression of an integrated data model for Web Services must be an XML data model. In other words, the constraints for message payloads are captured in XML Schema. To enable multiple teams of consumers and developers to participate in the SOA, schemas are externalized, standardized, and federated.

1.1. Externalized, Standardized, Federated

Let's examine the terminology before proceeding to the technology. An ‘externalized’ schema is a schema that describes the message payload of a Web Service, but is not contained in the body of the WSDL that describes the technical implementation of the service. Rather, the WSDL refers to an externally available schema file. This helps to insulate the service from maintenance when minor changes are required to the payload. An extra benefit is to expose an exact contract to any developer who needs to process the data described by the schema.

‘Standardizing’ schemas means conforming to a set of agreed standards for all object definitions and constraints – for example, the format of an address, the names of highly used data objects. Often, XML taxonomies from trading partners and vertical standards bodies strongly influence this standardization process.

‘Federating’ schemas means distributing schemas – on a ‘pull down’ or ‘push’ basis, either is possible – to all consumers and stakeholders. Registries should support the federation of schemas and associated assets (transformations) to communities of consumers (the ebXML Registry specification provides a good example of this). Essential to managing this process is an agreement that schemas and associated assets are centrally managed. If the SOA is to remain stable, modifications should not occur in the body of federated schemas and transformations, rather, modifications belong in the model that was used centrally to generate the schemas in the first place. This is the price of ensuring that all the various parts of the SOA remain synchronized.

The relationship between federated schemas for payloads and the underlying application models is managed through transformations, which are also built against the integrated data model. These transformations form the interface layer, where data is transformed from its format as described by the integrated data model into the format required by the application model, and vice versa. A transformation is effectively an expression of a relationship between two schemas (and a schema for a service payload must therefore be seen as a view on a model). These transformations are built against the integrated data model, which must therefore have knowledge both of the underlying data models and of the payload schemas that have been assembled from the model.

RenderX
XSL • FO
formatter

Re-format page sizes

1.2. Active Metadata

When a layer of externalized schemas is created to constrain message payloads, the metadata represented by those schemas takes on an active business role, as opposed to a passive technical role. That is, introducing a change to a business process or object moves the onus of development and maintenance work out of the code and into the metadata. A developer must first change the schemas (and associated assets such as transformations) before addressing the code. The way that we currently capture XML-metadata and its related transformation logic in deployable schemas and transformations, particularly when used and maintained by multiple consumers from multiple organizations, forces us to proliferate references to the same objects over and over again across the shared family of schemas and transformations. Such duplication of references is not to be confused with redundancy, because if that were the case we could apply better design practices to eliminate the redundancy. Duplication is an unavoidable price that comes from the file basis used to contain schemas and transformations.

1.3. Impact Analysis

With multiple developers working on schemas, and multiple consumers of the resulting schemas objects, it is essential to be able to predict the consequences of any given modification. Without impact analysis, predicting the impact of a given change and applying version control to a system as it evolves are both complex and largely manual tasks, with a high risk of error and potential for incomplete coverage in complex systems. This is because maintaining complex systems that depend on active XML-metadata layers, especially when there are multiple consumers and development teams collaborating on the system, forces developers to work in the schemas and transformations themselves, at the file level, exercising manual searches for object references and implementing what is often the same change repetitively on a case-by-case and file-by-file basis.

Post-development, when a schema-constrained system is live and the ‘code’ is in maintenance mode, the most difficult part of making any modification is knowing categorically that all the consequences of that modification have been charted and addressed. The modification should not cause any unpleasant surprises. What we actually want in order to understand the consequences of any given modification is a report that lists all affected objects and their context, with visibility to developer and consumer alike (or as appropriate).

Accordingly, we need to shift away from the file basis for our management of XML-metadata and associated assets. One of the primary goals of SOA is truly scalable flexibility and independence from underlying technologies. Our management of XML schemas must similarly become independent of underlying technologies through a shift to model-driven management and an automated generation of schemas and transformations. Only then can the metadata become active.

1.4. Necessary Layers of Abstraction

Staying with SOA for the a while yet as the argument in this paper unfolds, between Web Services on the one hand, and the underlying application interfaces on the other, there are at least three layers of abstraction and metadata: the schemas layer; the integrated data model from which all new schemas can be derived and where, ideally, all modifications are initially made; and the transformation layer. These layers are necessary to allow Web Services to remain loosely coupled and dynamic.

RenderX
XSL • FO
formatter

Re-format page sizes

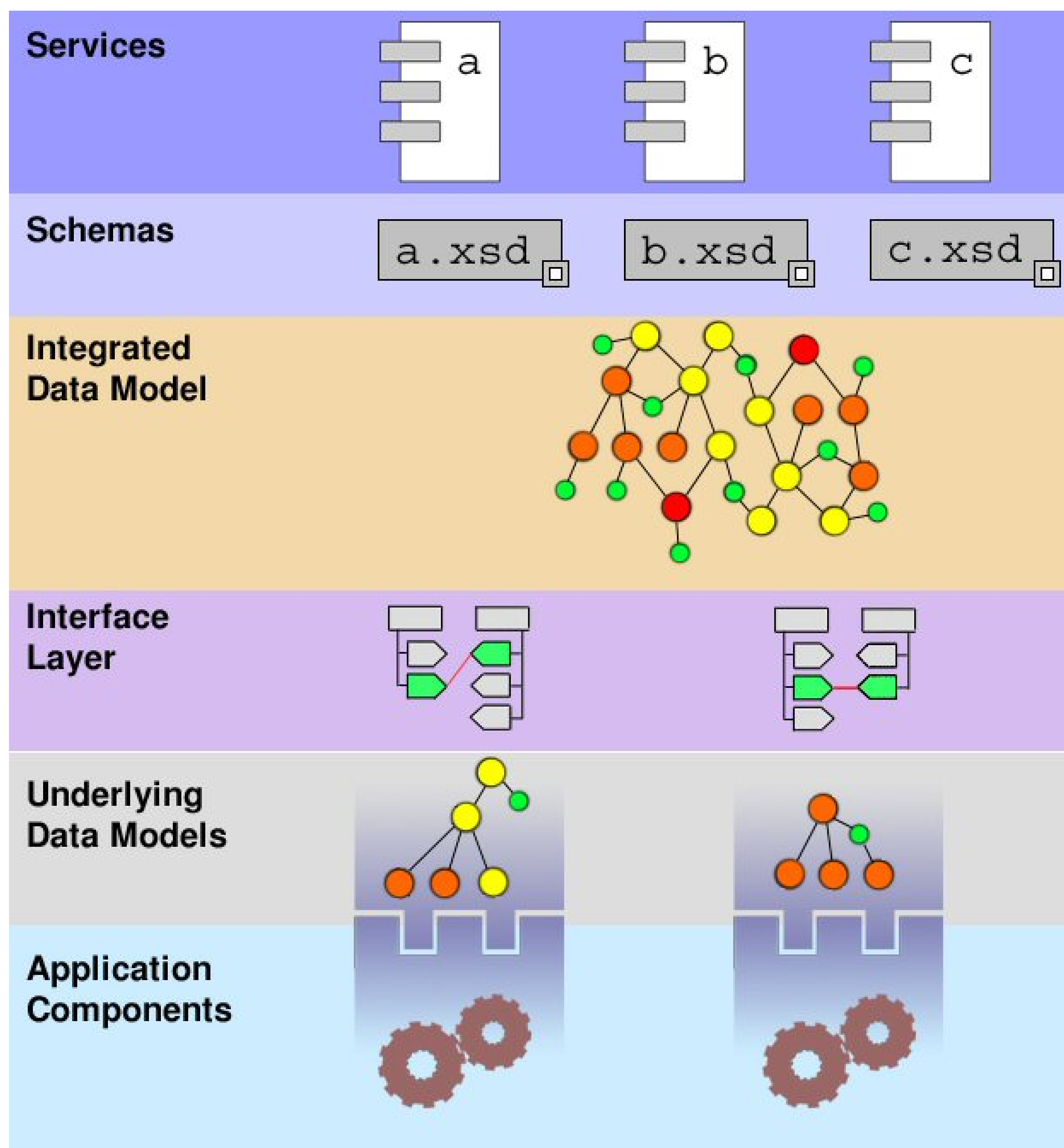

Figure 1. An Integrated Data Model, with Schema and Interface Layers.

The benchmark for testing whether services are sufficiently loosely coupled and dynamic is the level to which services are application-specific. It must be possible to unplug a given underlying application component and plug in an equivalent from another manufacturer, without updating the service. Conversely, different but equivalent services should be interchangeable without consequence.

RenderX
XSL • FO
formatter

Re-format page sizes

2. Fragility Through De-normalizing

An integrated data model should allow minimal redundancy and zero duplication. A conceptual view of the processes and data flows of an organization should therefore assume a model that, if implemented in a relational database application, would be normalized to the third normal form (see C.J. Date, The Database Relational Model: A Retrospective Review and Analysis, ISBN 0-201-61294-1, 2001 Addison-Wesley Longman, Inc.).

The expression of this model in XML Schema, with the intention of describing and constraining the payloads of Web Services, creates a layer of metadata that is anything but normalized to the third normal form. In a normalized model, changing the definition of, say, a customerID element, would require a modification in one place, namely the customerID object. This duplication occurs in XML schemas because schemas express metadata, but are essentially not models. They are deployable expressions of a model, for use with validators, parsers, and so on. What we are missing in this picture is the model of the schemas.

As we have seen above, the immediate repercussion of duplication of references across schema families is the effect it has on your ability to predict the impact of a change to an object in the model. In a family of schemas and transformations shared across multiple development teams, there is no robust, automated way of identifying firstly where any given object has been referenced, and secondly what the impact will be of a modification to that object.

If we consider that our starting point is a 'normalized' model, our modifications should take place in the model, not in the deployed expressions of the model. However, for object-level version control and impact analysis to be possible, a record must be kept of all externalized expressions of the model in order for an association to be maintained between the version level of the conceptual object and the instances of its use in a system. It must be possible for a developer to safely work on a metadata model, incrementing object versions for all modified objects, without worrying about the task of synchronizing existing deployments with the modifications. Implicit in this requirement is the need for an engine that can regenerate all externalized expressions of the model upon demand. Modifications are therefore model-driven.

3. Model-Driven Metadata Management

This paper deliberately avoids advocating the application of a Model-Driven Architecture (MDA) as specified by the Object Management Group (OMG) (see MDA Guide Version 1.0.1 [http://www.omg.org/docs/omg/03-06-01.pdf], Editors: Joaquin Miller and Jishnu Mukerji, OMG 2003, Document number OMG-2003-06-01). Metadata management in an SOA implementation is too specifically XML-based for MDA to add much value once a system has gone live. At an early design stage, inevitably much use will (and should) be made of design methodologies and tools such as UML that have little to do with XML (at least, directly). However, when an integrated data model exists, and a schema layer and a transformation layer have been created, the dependency on XML schemas is too great to warrant switching out of the XML in order to model modifications in a more general, non-XML model.

3.1. Round-tripping Through a Non-XML Model

Various attempts have been made – for example, using XML Metadata Interchange (XMI) – to 'round-trip' through a UML model to drive the process of developing, deploying and maintaining schema families, with varying degrees of success. The most significant reason that these have not been widely adopted is that the granularity of the objects that can be individually addressed, versioned, and analysed for impact is too rough. Typically only schemas, or schema fragments, can be 'round-tripped', which is of little help when you need to address object-level modifications to elements, attributes, complex types, and so on.

Dave Carlson of Ontogenics Corp. presented a paper on a related subject at XML 2001 in Orlando, Florida, entitled "Integrating XML and Non-XML Data via UML, XML 2001 [http://www.idealliance.org/papers/xml2001/papers/html/05-00-02.html]". Provided that a UML model remains the master in the deployed environment, round-tripping of schemas through edit cycles is achievable. However, this does not meet a long list of specific requirements of a model to support XML schema management.

RenderX
XSL • FO
formatter

Re-format page sizes

3.2. Specific Requirements of a Model to Support XML schema Management

Model-driven metadata management for XML schemas and transformations requires the following for a sufficiently powerful implementation to succeed:

- A model that fully includes, and is a significant superset of, XML Schema;
- A model that is rich enough to allow mappings between objects in multiple schemas, and capture sufficient information to enable an engine to auto-generate XSLT;
- A model that can conceive of XPath in order to map object-to-object relationships correctly in any given context;
- Extensibility, to cater for the inclusion of related, non-XML assets in the deployed environment;
- Object-level granularity (to at least the simple type grain);
- Object-level version control;
- A workflow that starts with a ‘construction’ phase to create and modify objects, followed by an ‘assembly’ phase in which structures such as schemas are assembled from the pool of objects created in the construction phase, and a ‘deployment’ phase that uses a model-driven architecture to generate and export the structures assembled in the assembly phase;
- Importers that can burst schemas into their constituent objects and structure relationships;
- Listeners that can react to change events;
- Impact reporters with a ‘veto’ option to rollback modifications that have an unacceptable impact.

This is a largely conceptual list, with no implementation specifics. In order to satisfy these requirements in software, the type of model must be fixed – this is essentially a choice between object oriented and relational (see below) – and a proof of concept or prototyping study undertaken to test the ability of the software model to meet the functional requirements that we will make.

4. Functional Requirements

In order to support multiple developers working in multiple teams, version control and impact analysis are essential. The functional requirements of an XML schema management system that is capable of supporting version control and impact analysis can be sub-divided into the following categories:

- Import;
- Editors;
- User Access Control Layer;
- Version control;
- Export.

RenderX
XSL • FO
formatter

Re-format page sizes

4.1. Import

A model for XML schema management must be capable of importing existing schemas and the output of equivalent models. A minimum requirement is to be able to import XSD and DTD formats. This is essential because very little schema development these days uses a clean sheet as the starting point. One needs to be able to import existing schemas. The file-based packaging of schemas makes it inevitable in large distributed environments that multiple references will exist to individual objects.

During the import process, it must be possible to resolve issues of duplication and conflicting definitions according to namespace constraints. That is, when importing an object in a given namespace that the system recognizes as already existing in the model, it must be possible for an operator to take a decision on the basis of the following choices:

- Replace existing;
- Create copy;
- Ignore.

Namespaces must be correctly declared.

The requirements for importing and bursting XML Schema files are as follows:

- When importing a schema that includes another schema, the importer should check that the namespace of the included schema matches the namespace of the including schema;
- Global element, notation, model group, and type declarations should be imported as top-level objects in the tree;
- Local declarations of elements, attributes, model groups, notations, and types should be imported as children of their parent object;
- Element cardinality should be held by the model group;
- Simple content and complex content declarations can be safely ignored, as this information can be derived;
- Importing a schema that uses <redefine> to change an object declaration via redefinition should cause the importer to create in the model both the original declaration as well as the replacement declaration. This is because the concept of <redefine> is cumbersome to express in a single-source object model and is therefore problematic to display and manage reliably in a tree view. A human-being must subsequently choose to keep the replacement or the original declaration when the import action is complete.

When importing a DTD:

- All element, notation and general entity declarations should be imported as top-level objects in the tree;
- If an attribute list occurs without a corresponding element definition in a DTD, then an element definition should be created to hold the attributes;
- All attribute declarations should be imported as local (element-level) definitions;
- Parameter entities should be expanded;
- Content models should be imported as anonymous local (element-level) model groups;
- Elements with child elements or attributes should be imported as having local complex types (complex types are not defined as such in DTDs). Global complex types should not be created;

RenderX
XSL • FO
formatter

Re-format page sizes

- All cardinality should be mapped to model groups, in order to scale up to XML Schema conventions. For example, an optional element should be imported as a mandatory element inside an optional model group (so `x?` is imported as `(x)?`). This change helps to improve re-usability within the XML object model;
- Comments should be converted to annotations;
- It should be possible for a human-being to influence the logic governing which object newly created annotations are attached to when importing comments;
- Contiguous groups of comments should be merged into a single annotation;
- Any comments at the end of the DTD should be attached to the top level of the tree;
- Elements containing #PCDATA should be given a complex type with a base type of String.

4.2. Editors

A model for XML schema management must be capable of opening any individual object in a tree for editing, and creating new objects as required. The 'Delete' action must also be supported. When editing an object, it should not be of any consequence whether the user has accessed the definition of an object or a reference to the object. The system must be able to locate and offer the source definition for editing.

Editing an object should happen with no regard to the schemas and transformations that contain or reference that object.

Editing should be confined to objects only (elements, etc.) and not containers (schemas, etc.). That is, a deployable object in XML-metadata terms is always a container, such as a schema, and should be assembled from editable objects. This is essential if we are to support version control and impact analysis at the object level, rather than the currently rough-grained schema-level. It is also essential for supporting the assembly of schemas by non-technical personnel when they wish to modify or extend application behavior in an SOA.

The Construct—>Assemble—>Deploy workflow that is applicable here makes it possible to carry out true single-source editing of objects in a model, and to assemble them limitlessly into schemas and transformations without fear of duplication or redundancy. Also, having assembled objects into deployable structures such as schemas, and having stored a record of that assembly in the model, we can predict the impact of modifications to schemas from the perspective of the object being edited.

Container objects in XML (schemas, transformations) have properties that determine, for example, how to interpret namespaces at run time, and these properties must be available for editing, which means that it must be possible to create multiple deployment records for deployable objects, in order to cater for variations.

When editing any objects, the system must be able to analyse and report the impact of the modification. In a tree based representation of an object model, impact analysis works with back references from objects in the internal tree structure of the model. That is, given a change event, the impact analysis tool should walk the back reference tree from the given object, collecting relevant objects to be reported. The resulting set of objects is then given to the impact listeners that are registered with the current session. Any of these impact listeners can 'veto' (prevent) the change. The ensuing rollback makes a transaction management framework essential. When a change event happens, the impact analysis tool should filter the results so that only relevant impacted objects are reported to the impact listeners.

Which objects are reported for impact in any given modification is open to much discussion. Not only do opinions differ on what 'impact' actually means, but the list of impacted objects that development organisations will expect to see for a given modification is invariably different for every organisation. A safe guideline for standard XML objects managed by the model is to assume that the following are affected by a modification at object level:

- The first named top-level attribute, attribute group, model group, complex type, simple type, notation, and entity before the first element;

RenderX
XSL • FO
formatter

Re-format page sizes

- The first element;
- The first schema reached via node references (not schema references);
- Object-to-object mapping instances;
- Schema-to-schema mapping sets;
- Custom objects that extend the model.

For example, in the following illustration, the change event is triggered on a modification to element E (the top level model group is being changed from a sequence to a choice). Element E is contained in schema S2. Schema S1 references schema S2. To analyse the impact of this change, a back reference tree walker is set off from the sequence, walking up the tree. Element E is reported because it is the first element reached from the sequence. Schema S2 is reported because S2 contains element E and is thus marked as modified, which means that all schema instances created from this model-level schema are no longer valid and must be redeployed. Schema S1 should probably not be reported, because schema S1 is not directly affected by this modification and therefore does not need to be redeployed.

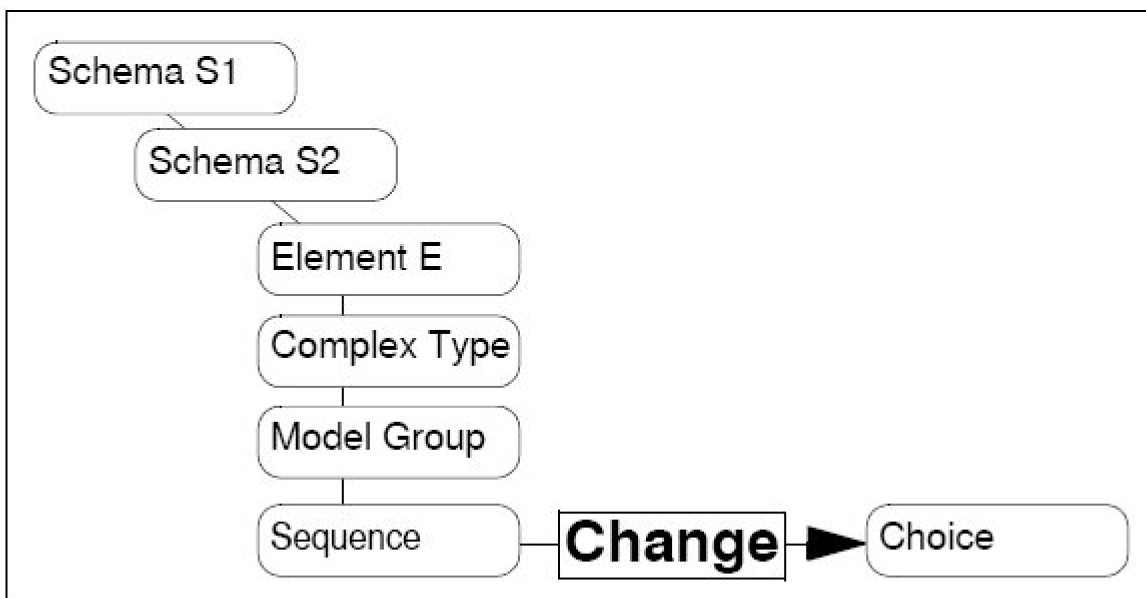

Figure 2. Analysing the impact of a modification.

4.3. User Access Control Layer

Version control and impact analysis in XML schema development projects only become a serious issue when there are multiple developers or development teams collaborating on and consuming the families of schemas and transformations. When one person controls the schemas used in a SOA, applying version control is a relatively straightforward exercise, and analysing the impact of change is not particularly difficult.

When multiple developers access the model, a user access control layer must be applied. In the user access control layer, provision is needed for users, user groups, user roles within each group, and permissions that can be applied to roles. Furthermore, a locking system and transaction management must be applied, to prevent parallel updates at critical moments. A communication layer must also exist to inform user groups of important events, such as a new release of the schemas.

RenderX
XSL • FO
formatter

Re-format page sizes

4.4. Version control

To manage version control effectively across multiple users, there must be a mechanism for checking out objects from a central repository, and an equivalent mechanism for checking modified objects back in to the central repository.

When multiple developers work on a shared set of source objects, conflicts can occur because different modifications might be made to the same object in separate sessions. These conflicts fall into the following categories:

- ‘Version’, when multiple developers modify the same object in separate projects and check in their changes;
- ‘Remove’, when an object has been removed by one developer and the same object exists in a modified state in another developer's checked in work;
- A ‘one-to-one link’, where only one link is permitted (for example the name of the complex type that is applied to an element) and the same object has been modified in separate sessions to point to different targets;
- A ‘one-to-many link’, where a link is allowed to multiple other objects (such as a list of attributes) that must conform to a constraint (usually uniqueness) that is violated in two separate sessions.

In all cases, a conflict only arises in cases of parallel changes to the same object. This implies that both the developer (that is, the person checking in the work), and the administrator (the build manager integrating the checked in work) must be given the chance to resolve conflicts when they are detected, or back out of the update/integration action.

The results of an integrated set of checked-in work is a build. That is, the model *in toto* is recognized to be at a new version level, containing an incremented version identifier for every modified object that it contains relative to the previous build.

Ideally, an administrator should be able to promote any previous build to be the current build, in order to enable the opening of previous versions of the model for editing.

4.5. Export

The results of model-driven XML schema development are schemas and transformations, produced in an identifiable, version-controlled release. This means that any release defined for the model must be associated with one build only. A release takes all the assembled deployable structures (schemas and transformations) and prepares them for the release according to parameters such as the choice of schema implementation (xsd or dtd), providing an administrator with records that can be edited for deployment parameters such as the defaults for namespace behavior.

In accordance with model-driven principles, creating a release should not be the same as deploying a release. Rather, the export functionality needed to generate schemas and transformations should be invoked when an administrator chooses to federate the results. At that point, software generators driven by the object and structure definitions in the model create and export schemas and transformations in an entirely repeatable manner.

5. Automating Assembly

This paper has described the technical and organisational challenges facing multiple-developer schema development projects, particularly when those developers represent multiple departments or organisations. The infrastructure that is necessary to address these challenges provides a truly single-source object modeling environment for XML. With a single-source object modeling environment, it becomes possible to assemble schemas and transformations from sets of objects without concern for where those objects are used. Thus it is possible to create a ‘fast-track’ implementation of business rules and processes, as it were. For example, when a Web Service is needed to process a credit check on a new customer, a business-focused engineer with no knowledge of the finer points of grammar and syntax should be able to create the required payload schema for the Web Service from a series of existing objects: the customer details,

RenderX
XSL • FO
formatter

Re-format page sizes

the bank details, and so on. Hey presto, another schema, together with automatically generated WSDL, XSLT templates, and technical documentation!

6. Summary

Version control and impact analysis in XML schema management and development are essential enablers of any collaborative schema development effort across multiple, distributed teams of developers. For version control and impact analysis to be sensibly feasible, we must work from within the perspective of a single-source object model. Development and deployment of schemas and transformations must be model-driven – that is, driven by modifications to active business metadata from the perspective of the model.

The granularity of the model must be truly object-level, not schema-level. It must be possible to generate deployable schemas and transformations from the model in such a way that developers no longer need to implement changes by editing schema instances and transformations.

In conclusion, especially when applied to a SOA, this approach will give us a considerably more powerful metadata management layer between Web Services and underlying application components.

Bibliography

C.J. DateThe Database Relational Model: A Retrospective Review and AnalysisISBN 0-201-61294-12001Addison-Wesley Longman, Inc.

Joaquin MillerJishnu MukerjiMDA Guide Version 1.0.1Document number OMG-2003-06-01

Dave CarlsonIntegrating XML and Non-XML Data via UML2001XML 2001 (Orlando, Florida)

RenderX
XSL • FO
formatter

Re-format page sizes

Biography

Jim **Gabriel**
digitalML Ltd. [http://www.digitalml.com]
Centurion House
London Road
Staines
Middlesex
TW18 4AX
United Kingdom

Jim Gabriel is an inventor who has lived through a number of metadata evolution management problems. Director with London-based digitalML Ltd.

RenderX
XSL • FO
formatter