Open Applications Group
Best Practices and XML Content for Everywhere-to-Everywhere Integration

OAGIS 8.0
Design Document

Authors:
Michael Rowell,
Mark Feblowitz

Editors:
Kurt Kanaskie
Mark Feblowitz
Andrew Warren
Michael Rowell
David Connelly

Document Number: 20021024-1

NOTICE

The information contained in this document is subject to change without notice.

The material in this document is published by the Open Applications Group, Inc. for evaluation. Publication of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, OPEN APPLICATIONS GROUP, INC. MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANT ABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Open Applications Group, Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

For more information, contact:
Open Applications Group, Inc.
1950 Spectrum Circle, Suite 400
Marietta, Georgia 30067 USA
Telephone: 1.770.980.3418
Fax: 1.770.234.6036
Internet: http://www.openapplications.org

Table of Contents

OAGIS 8.0 Design Document

Abstract

This document discusses the design decisions that have been made for OAGIS 8.0; this includes the design of OAGIS 8.0 itself and the instantiation of OAGIS 8.0 in XML Schema.

As a widely used eBusiness and Application Integration specification, OAGIS has evolved and matured since 1996. OAGIS has grown from a relatively small set of specifications for financial transactions to a diverse, canonical, horizontal specification consisting of approximately 200 BODs.

As OAGIS has grown and matured, practices have changed and "issues" have been identified. The design of OAGIS 8.0 – and its instantiation in XML Schema – together represent a significant modernization of OAGIS. OAGIS 8.0 has been designed to address many of the shortcomings of prior OAGIS instantiations, while preserving and building upon the strengths of the OAGIS specification.

Changes incorporated into OAGIS 8.0 include: improved naming conventions, removal of content ordering restrictions, resolution of the "nondeterministic" problem, adoption of current "best practices" in XML modeling and representation, improvements to UserArea extensibility, and the addition of overlay extensibility, which allows users to extend OAGIS to create industry/vertical overlays and company-specific adaptations.

This document identifies the key issues addressed by OAGIS 8.0 and describes the design principles, architecture scope, and additional content that are present in OAGIS 8.0.

1.0 OVERVIEW

As a widely used eBusiness and Application Integration specification, OAGIS has evolved and matured since 1996. OAGIS has grown from a relatively small set of specifications for financial transactions to a diverse, canonical, horizontal specification consisting of approximately 200 BODs. As OAGIS has grown and matured, practices have changed and "issues" have been identified. OAGIS 8.0 has been designed to address many of the shortcomings of prior OAGIS instantiations, while preserving and building on the strengths of the OAGIS specification.

Some of the shortcomings of OAGIS stem from the evolution process itself – as new BODs were added over time, new content was discovered and added. Older BODs that carried related content were not always updated, since they were already deployed and functioning. This has led to a degree of inconsistency among related BODs, BODs that should be carrying the same content.

However, most of OAGIS' current shortcomings are due to design practices and technical limitations that were in place during OAGIS' early history – naming conventions, architectural practices, XML representation conventions, etc. A major portion of the learning curve for OAGIS is devoted to learning how to represent the simplest things – dates, times, quantities, etc. Most of these challenges are due to limitations of XML 1.0 DTDs, OAGIS' prior instantiation base. The inability to easily add content to related BODs was the direct result of inadequacies in how DTDs represent reusable types.

The design of OAGIS 8.0 – and its instantiation in XML Schema – together represent a significant modernization of OAGIS. This design endeavors to repair many longstanding problems with prior OAGIS instantiations, and to normalize many of the related BODs that were designed at different times and with inconsistent representations.

The goals for the OAGIS 8.0 redesign include:

1. ***Modernize* OAGIS** - The need to modernize the design of OAGIS as indicated by End Users and Solution Providers feedback.

2. **Address the "Non-Deterministic Issue"** – The need to strengthen the XML validation of the correct structure and content of an OAGIS model (which was primarily due to the weak typesystem available in XML DTDs).

3. **Address requests for long-tag names** – A common practice at the time OAGIS was originally designed was to use short tag names to reduce the amount of overhead imposed on the message. Today, most applications of XML make use of unabbreviated tag names. OAGIS has been criticized for not modernizing the legacy-based all-uppercase, heavily abbreviated names.

4. **Provide better vertical extensions capabilities**; address the inability to cleanly represent particular verticals' overlays onto OAGIS - While it is possible to extend OAGIS, these extensions have been relegated to the USERAREA. This leaves the user of a vertical overlay with the impression that the verticals' extensions are not part of the specification.

5. **Support XML Schema** – OAGIS needs to support W3C's XML Schema Recommendation (2 May 2001), and to use the expanded representational capabilities to address some of the key representational issues in OAGIS.

When polled, the constituency of the Open Applications Group felt that these issues must be addressed in order for OAGIS to continue to be recognized as a leading specification for integration in industry.

This modernization is necessary to enable OAGIS users to efficiently deploy OAGIS BODs and to extend OAGIS to create industry/vertical overlays and company-specific adaptations.

2.0 OVERALL DESIGN

The following are the design principles, architecture scope and deliverables for the design of OAGIS 8.0.

2.1 Design Principles

These design principles are core to what has made OAGIS successful over the years and key to how OAGIS will continue to be successful.

1. No feature regression!
2. Keep it simple!!
3. Retain existing conventions where possible for end user familiarity
4. Add support for XML Schema, but stay away from ambiguous features of XML Schema
5. Retain independence of OAGIS from the instantiations of OAGIS
6. Retain and expand extensibility
7. Limit new application functionality
8. Define migration path for users.

Many see OAGIS' saving grace as its ability to be extended to meet the needs of the customers, vendors, and trading communities that make use of it. For this reason, it is critical that OAGIS maintain it's Extensible Content Model and where necessary expand upon it in order to facilitate the requirements of the vertical industries, without breaking the return-on-investment of having a canonical content model that an enterprise can use.

2.2 Changes from OAGIS 7.x

The primary goals for OAGIS 8.0 are to address the issues raised in the previous section while adhering to the above design principles, making use of prevailing standards where applicable.

1) Address the "non-deterministic issue." (see section 3.4 Address *Non-Determinism)*

2) Use long tag names.
   a) Use the OAGIS long names currently listed in Appendices C and D. An example of this would be to replace ACCTTYPE with AccountType
   b) Change capitalization to ISO 11179 – Upper CamelCase for elements, lower camelCase for attributes.
   c) Tag names may be any length but require OAGI technical approval for any name over 31 characters.

3) Convert to XML Schema (see section 5.0 XML Schema Instantiation)

4) Change the name of the Control Area to ApplicationArea
   a) Eliminate Code Page tag from ApplicationArea
   b) Eliminate Language tag from ApplicationArea
   c) Add Digital Signature to ApplicationArea

5) Acknowledging that not all parties to a transaction are engaged as partners, change Partner to Party. Create new constructs to represent Party information and Party types, and to refer to a specific Party.

6) Support other standards where applicable
   a) This includes the use of ISO standards, where applicable. For example use of ISO DateTime format
   b) Others, where applicable

7) Update the use of OAGIS terminology
   a) Data Types will become Components.
   b) Segments will become Compounds.

8) Improve the extensibility of the OAGIS UserArea.

9) Provide migration tools for customers. If feasible, use XSL style sheets to convert from DTD/XML to Schema XML.

10) Remove the pre-OAGIS 8.0 content ordering restrictions; instead, allow elements to appear in a Component in a more natural order.

11) Defect Removal

2.3 OAGIS 8 New Features

1) Provide the ability, where appropriate, to precisely type the values that occur within a field or attribute. In many instances it may be preferable to allow for flexibility for the values that may occur. This must be determined consistently, and thus requires OAGIS to establish well defined and repeatable representation practices for defining and applying types.

2) Provide an extensible equivalent to enumeration types; allow OAGIS extenders to provide additional values, e.g., Party types (CustomerParty, SupplierParty, ShipToParty, etc.), and have those values be validated by an XML Schema validating parser.

3) Provide verticals with the ability to define extensions to OAGIS such that the added content appears as sibling elements to OAGIS-defined fields, compounds, and components (not just relegated to a UserArea).

4) Prevent name collisions and "naming bloat"[1] by using namespaces to distinguish among names from different domains (names in OAGIS, in and overlay, in a company-specific extension, etc.).

5) Provide a mechanism to maintain a consistent representation of the Noun content across a family of BODs. That is, any content that is available in one specific noun, e.g., PurchaseOrder, is available in all PurchaseOrder BODs (ProcessPurchaseOrder, SyncPurchaseOrder, CancelPurchaseOrder,…). Specifically, this is achieved by creating a full definition for each Noun (everything that any PurchaseOrder might contain), and allowing the unique

[1] Namespace bloat is where simple names, e.g., Address, are pre/post-pended with disambiguating information, e.g., STARDeliveryNeedByDateTimeAny

requirements for each BOD (each PurchaseOrder BOD) to be expressed as XSL-encoded constraints. These constraints can be enforced by any standard XSL processor.

6) Provide a mechanism whereby elements of similar type share content in a uniform way; for example, where all Document share the same Document identifiers, etc., where all Orders share the same kind of order-specific content (unit price, total price, description, item identification, etc.). This is achieved by recognizing Nouns, etc., of the same type and defining a common type base that is to be shared among the elements.

7) Factor Verb content from Nouns; verb-specific Noun content (verb-oriented parameters, etc) is now represented as a part of the Verb, simplifying Noun definitions.

8) Add the ability to provide vertical extension ***overlays***. This will allow a vertical to add additional information that provides additional information or detail for their overlay. These overlays will be defined through continuing to work closely with other industry verticals. These "Vertical overlays" will build on the OAGIS XML Schema instantiation and be provided as separate namespaces in order to distinguish new elements and element types.

9) In order to take advantage of the advanced typing system available in XML Schema and with the desire to keep these extensions simple OAGIS 8.0 will make use of XSL to provide runtime constraints of the required fields. This allows OAGIS to be extended easily while taking advantage of a very reach and flexible constraint mechanism. Without adopting a completely open content model.

10) Maintain an acceptable level of performance for use of the BOD .xsd files (XML Schema Definition files) with the dominant XML tools available on the market.

3.0 UPDATES TO OAGIS

In order to achieve the scope identified in the previous section, while following the design principles provided. The following updates will be made to OAGIS:

3.1 Naming Conventions

In order to be consistent with current practice the Open Applications Group has adopted full (long) tag names, in accordance with the long names documented in Appendices C and D of OAGIS 7.2.1. As an exception to the use of long tag names, a small number of widely accepted abbreviations have been adopted, e.g., Id for Identifier. In general, readability and consistency are best preserved when abbreviations are kept to a minimum.

OAGIS 8.0 element, type, and attribute names have been capitalized in accordance with ISO standard 11179, using *UpperCamelCase* for elements and *lowerCamelCase* for attributes. Examples of these long element names are: EffectivePeriod, and AccountType. Examples of the long attribute names are: entryDateTime, owner, etc.

For abbreviations the naming-convention is as follows:

For element names and type names:

- The first letter of each abbreviated word should be capitalized
  - UnitOfMeasure will be abbreviated UOM
- The second and subsequent letters of each abbreviated word should appear in lower case
  - Identifier will be abbreviated Id
  - Indicator will be abbreviated Ind

For Attribute names:

- All abbreviated words at the beginning of an attribute name would appear in lower-case.
  - UnitOfMeasure would be uom (uOM just doesn't look right).

3.2 Update OAGIS Restrictions

Update OAGIS to remove restrictions that are no longer needed or to redefine restrictions where warranted. These include:

1) No longer require that every field or compound be present in the ApplicationArea. The only required element in the ApplicationArea is the Creation DateTime.

2) Remove the OAGIS restriction to have the elements appear in a certain order within a Component, but instead allow them to appear in a more natural order. Pre-OAGIS 8.0 ordering restrictions are removed: segments/compounds no longer need to precede fields; segments/compounds no longer need to precede optional segments/compounds; required fields need not precede optional fields; no alphabetical ordering is imposed.[2]

3) Remove the restriction that all fields defined within a segment/compound must occur for each occurrence of the segment/compound. This will allow for useful/meaningful subsets, which can be explicitly defined or restricted using OAGIS content constraints.

3.3 Update Constructs and Terminology

As indicated in the *OAGIS 8.0 Design* section of this document, OAGIS 8.0 will update some of the OAGIS constructs and terminology:

1) The Control Area has been renamed to the Application Area; as a result the CNTROLAREA tag has been changed to "ApplicationArea".

2) The ApplicationArea has been made generic and uniform across all BODs. Any BOD-specific content that once appeared as a part of the CNTROLAREA has been relocated to the BOD element itself (as attributes) or to the respective Noun or Verb element in the DataArea.

[2] Note that once a particular ordering of fields and/or components has been defined in the schema, ordering of content within the XML instance must correspond to the schema definition.

a. The revision number has been removed from the top-level element of OAGIS and replaced by a “revision” attribute on the top level BOD element (see the OAGIS definition of the BusinessObjectDocument for a definition of what comprises a BOD).

b. An “environment” attribute has been added to the top-level element of the BOD with the possible values of “Test” or “Production”.

c. The BSR segment has been removed from the ApplicationArea. This information is already captured both in the name of the BOD element and in the names of the BOD's Verb and Noun elements, respectively.

3) The Language and Code Page fields have been removed from the Sender segment. These fields are no longer useful since XML allows the user to identify a language. This is accomplished as described at: http://www.w3.org/TR/2000/WD-xml-2e-20000814

4) A “Signature” element has been added to the ApplicationArea in order to enable digital signature of the entire BOD contents. The Signature element allows a digital signature to be associated with a BOD instance. There are several specifications for Digital Signatures, the Signature element here can be used to carry any of these.

5) The DataArea now contains elements that identify the BOD's specific Verb and Noun. For example, the first element in the DataArea of the "ProcessPurchaseOrder" BOD will be the "Process" verb element and the second element will be the "PurchaseOrder" noun element. The Verb element defines or constrains the action of the BOD, and the Noun element carries the data.

6) A “UserArea” has been added to the ApplicationArea.

7) ISO DateTime format has replaced the existing DateTime structures.

Due to theoretical limitations and practical considerations, XML Schema does not allow for the

8) The OAGIS term *segment* has been renamed to ***compound***. A compound is a set of elements and attributes that can be thought of as one atomic concept, e.g., Amount. Compounds are not extensible via the OAGIS extensibility methods.

9) The OAGIS term *data type* has been renamed to ***component***. Components are the large grained building blocks that are used to construct/compose Nouns. They group Fields, Compounds and other Components into extensible groups.

3.4 Address Non-Determinism

Non-determinism can roughly be defined as a situation where, upon encountering an element in an instance document, it is ambiguous which path was taken in the schema document.

Ninety percent of the instances of OAGIS non-determinism occur with how earlier versions of OAGIS segments were represented, due mostly to limitations of XML DTDs. A deeper explanation of this problem's basis in type theory is beyond the scope of this document. Suffice it to say that element non-determinism has been a thorn in the side of many OAGIS users.

3.4.1 The Non-Determinism Problem in a Nutshell

In prior versions of OAGIS, fields that relied on segments were named based on the ***intended type*** of a field (e.g., "DateTime"), not based on the ***actual name*** of the thing being described (e.g., "NeedDelivery"). What would have been the natural name of the field was instead buried in a "qualifier" attribute. So, instead of modeling the NeedDelivery field of a PurchaseOrderLine as

```
<PurchaseOrderLine>
        …
        <NeedDelivery>…</ NeedDelivery>
        …
</PurchaseOrderLine>
```

arbitrary ordering of element content in the instance.

it was modeled as

```
<PurchaseOrderLine>
        …
        <DateTime qualifier="NeedDelivery">…</DateTime>
        …
</PurchaseOrderLine>
```

This was one of the few ways that DTDs could impose the needed DateTime structure on the NeedDeliveryBy field, so that parsers could do some (minimal) checking of the content.

The problem arose when more than one field of type DateTime was needed in a given element model (e.g., more than one DateTime child of a PurchaseOrderLine):

```
<PurchaseOrderLine>
        …
        <DateTime qualifier="NeedDelivery">…</DateTime>
        …
        <DateTime qualifier="PromisedDelivery">…</DateTime>
        …
</PurchaseOrderLine>
```

The non-determinism exists because there are two different DateTime elements in the content of the PurchaseOrderLine . When the parser sees this and can't distinguish one from the other, it raises this as a warning. Furthermore, since the parse cannot distinguish one from the other, there is no way for it to require that, e.g., a NeedDelivery is required and a PromisedDelivery is optional.

The outcome of this is that, prior OAGIS 8.0, OAGIS designers were limited in what they could express in a given element, and XML parsers were limited in what structural integrity they could enforced.

3.4.2 Addressing the Non-Determinism

The problem is addressed by promoting the qualifier's value to being (part of) the element's name, e.g.,

<NeedDelivery>…</ NeedDelivery>

and by defining the element's model (type).

<element name="NeedDelivery" type="DateTime">…</element>

Now, rather than naming elements according to their types, elements are named according to their primary meaning, purpose, or function. Thus, there will no longer be an Amount(Extended)(T). Instead, the element will be named something like a required "TotalPrice" of type "Amount."[3] Furthermore, there can also be an optional "AdditionalCost" of type "Amount."

With XML Schema's relatively advanced type system, the context of the TotalPrice element and the binding, in the schema, of TotalPrice to the type Amount is all that are needed for a validating parser to validate that the content of a TotalPrice element is indeed an Amount and fits all of the criteria to be a legal Amount. Parsers can not only distinguish between a TotalPrice and an AdditionalCost, but can enforce that the former is required and the latter is optional.

[3] In all prior OAGIS releases, the practice of shortening field and segment names resulted in names that were less meaningful that their full equivalents, and often resulted in names that were inconsistently abbreviated. OAGIS 8.0 instead uses the long names that have long been associated with each element, as documented in Appendices C and D. For example AMOUNT(ESTFREIGHT)(T) in previous releases of OAGIS now uses the intended names, e.g., EstimatedFreightCharge.

4.0 DESIGN CONSIDERATIONS FOR OAGIS 8.0 IN XML SCHEMA

As of May 2001, the W3C released its first recommended version of XML Schema (W3C XML Schema Recommendation – 02 May 2001). Many XML users have eagerly awaited Schema's emergence, as did the Open Applications Group.

Overall, an XML Schema document offers a richer, more expressive, and more practical means of defining and constraining XML 1.0 instance documents than did its predecessor, the DTD (Document Type Definition). Many of the extreme challenges that OAGIS users faced when trying to represent their business documents in a natural way were due to the limited capabilities of DTDs, and the tight restrictions that DTDs imposed on XML instance documents.

As such, OAGIS is committed to remedying these difficulties by using the advanced expressive capabilities of XML Schema. This section describes OAGIS' use of XML Schema in representing BODs and supporting data, and describes the additional capabilities made possible via Schema.

Note that, as a first recommendation, XML Schema will likely evolve. The May 2001 recommendation is a stable, fixed definition, but the potential for improvement exists and future recommendations are expected. In fact, at the time of this writing, the W3C is currently working on XML Schema 1.1.

While XML Schema 1.0 represents a significant advance over DTDs, there are parts of it that can also be challenging to use, and there are a few areas where the Schema tool vendors do not fully agree on a joint interpretation of the recommendation. The Open Applications Group has endeavored, to the best of our understanding, to avoid these areas, in order to provide a more stable specification. The Open Applications Group will track XML Schema as it develops and will adopt new XML Schema best practices and additional core XML Technologies as they emerge.

Although XML Schema represents a significant advance beyond DTDs, XML Schema provides neither a complete nor a manageable way to express all of the necessary type constraints. Several months of

investigation and experimentation revealed that several of OAGIS' key design goals were neither fully achievable nor fully enforceable within version 1.0 of the XML Schema Recommendation. Until such capabilities become available in XML Schema, other core XML Technologies are being employed to address this limitation. XSL, in particular, supports the definition of schema constraints (using standard XPath expressions) that can be checked using any standard XSL processor; this capability can be used in conjunction with XML Schema validating parsers, resulting in a much more comprehensively screened set of BOD instances. OAGIS 8.0 relies on the use of these mature and standard technologies to provide additional validation support beyond that which is provided in version 1.0 of the XML Schema Recommendation.

OAGIS 8.0 takes advantage of the strengths of each of these core XML technologies: XML Schema to define the OAGIS grammar and XSL to define and enforce the constraints that strengthen that grammar. Together, these technologies offer a sound basis for representing and enforcing the core OAGIS standard, while also supporting the type of extensibility that OAGIS users require.

4.1 XML Schema and Types

A key difference between XML DTDs and XML Schema is XML Schema's advanced type system. In essence, everything defined in an XML Schema makes use of some kind of type definition, whether it's a built-in ***simple type***, a user-defined ***complex type***, or a modification of either (by ***extension*** or ***restriction***). Some of the user-defined types are ***anonymous*** – these types are unnamed and apply to only one element – while others are ***global*** – these are named types that are widely (re)usable, and even extensible or refinable.

XML Schema provides a rich set of built-in standard types (*simple* types), covering a breadth of common data types (strings, Booleans, dates, positive integers, etc.), many based on ISO standards. The existence of these built-in types (and the ability to validate that they are being adhered to) means that you don't have to create your own definitions for commonly used types, and that your applications can count on the data in a type-constrained attribute or element as being of the defined, commonly-used type.

By defining an attribute to be of type "xs:dateTime" – that is, the ISO 8601 "dateTime" type as defined in the XML Schema ("xs") namespace – your application can use a validating parser and then can count on the fact that the value of the attribute (e.g., 1999-05-31T13:20:00.000-05:00) will conform to ISO 8601. In addition to being a compact, standard representation for dateTime, most programming languages know how to parse the ISO 8601 dateTime, into month, day, year, etc., without requiring any custom-developed code.

In addition to XML Schema's built-in types, you can also define your own types, and you can derive new types from other types, either by *extension* or *restriction*. This is a very powerful mechanism; it allows a user to create types and build upon them or to restrict their usage as needed.

The existence of a type system provides significant expressive power that was lacking in DTDs. Many of the representation problems encountered when trying to represent OAGIS using DTDs find themselves easily resolved using the XML Schema type system. Perhaps the most important of XML Schema's improved capabilities is its greatly improved management of names. It uses namespaces to distinguish same-named concepts from different domains, e.g., boat:deck (from the marine domain, something that one stands on and sometimes gets wet) versus games:deck (from the games domain, a pack of cards). And it uses local names within a namespace to distinguish same-named parts of different things, e.g., a File's Owner, of type UserId, versus a Dog's Owner, of type Person. DTDs, on the other hand, operated in a completely flat namespace, requiring (very) specific names to distinguish each kind of thing, e.g., FileOwner or DogOwner. With the use of compact, local names, the XML over the wire becomes less verbose, easier to read and understand, and, in many cases, easier for applications to process.

OAGIS makes use of these capabilities in order to resolve many of the persistent modeling issues, and to provide a noun-based representation that promotes consistency across all of the BODs in a family of BODs.

4.2 OAGIS 8.0 and XML Schema Types

Because of the existence of a type system within XML Schema it is possible to use the built-in XML Schema ***simple*** types for many existing

OAGIS fields, and to define OAGIS types for other fields, compounds, components and Nouns. It is also possible to make use of XML Schema built-in types in place of existing OAGIS compounds for things like DateTime and to build more structured ***complex*** types for things like Temperature, Party, PurchaseOrders, etc. where needed. By making use of a type definition for each field, compound, component, and noun, it is possible to maintain the consistency of the fields and compounds across all uses of any particular noun or component.

4.2.1 Modeling of OAGIS BODs using XML Schema Best Practices

The XML Schema Best Practice guidelines from MITRE (authored by members of the xml-dev list and maintained by Roger Costello) recommends using what is referred to as the "Venetian Blind approach to XML Schema Design" when reuse is important. For a detailed explanation and a comparison of this approach against other approaches, see the paper at http://www.xfront.com/GlobalVersusLocal.pdf.

Simply put, the Venetian Blind approach emphasizes the creation of reusable types over the creation of global element definitions. So, rather than defining a global element called, e.g., "FileOwner" of type "SystemUser" and "DogOwner of type "Person", the Venetian Blind approach prefers that we define the (many) types that we need, e.g., the types FileOwner and DogOwner, that we derive the type FileOwner from type SystemUser and the type DogOwner from type Person, and that we ***bind*** the correct type to the element "Owner", depending upon the context. This allows Files and Dogs to have Owners, but ensures that the correct type of Owner is used in each context:

```
<File>
   <Owner userId="sysadmin">…</Owner>
</File>
```

and

```
        <Dog name="Rover">
            <Owner>
                <Name>
                    <FirstName>Mark</FirstName>
                    …
                </Name>
                <Address>
                …
                <Address>
            </Owner>
            <AnswersTo>Here, Fella</AnswersTo>

</Dog>
```

Clearly, in the different circumstances, a different type of "Owner" is warranted for each context.

If instead we use global element definitions to model all child elements (global element definitions must be uniquely named), we would have to define a FileOwner global element and a DogOwner global element and alter the File and Dog element models accordingly:

```
        <File>
            <FileOwner userId="sysadmin">…</FIleOwner>
        </File>
```

versus

```
        <Dog name="Rover">
            <DogOwner>
                <Name>
                    <FirstName>Mark</DogOwnerFirstName>
                    …
                </Name>
                <Address>
                …
                <Address>
            </Owner>
            <DogAnswersTo>Here, Fella</DogAnswersTo>

</Dog>
```

Taken to extremes, this would result in a very complex specification with very verbose and awkward naming conventions and a very bloated namespace (not unlike using DTDs).

In addition to aiding the management of names within a single namespace, the Venetian Blind approach also supports the use of multiple namespaces. This is highly relevant to OAGIS 8.0, since OAGIS 8.0 lives in a defined namespace, "http://openapplications.org/oagis" (with a typical namespace prefix of "oa:"), and since vertical and industry extensions to OAGIS 8.0 will live in their own separate namespaces

(e.g., namespace "AutomotiveIndustryXML" with prefix "aix:" or namespace "http://Cc:.com/oagis" with prefix "cc:"). Those curious about how Venetian Blinds aid in namespace management are encouraged to read the MITRE article.

OAGIS 8.0 adopts the modified Venetian Blind approach, making heavy use of type definitions rather than global elements. All of the compounds and fields, which were once global elements under DTDs, become elements that make use of types under XML Schema. Their exact instantiation within a BOD family can occur locally at the Noun. This allows OAGIS to locally define the elements based on the context of the usage within a given Noun family of BODs (that is, to locally bind the element to its appropriate type).

An extreme form of the Venetian Blind approach shuns the use of global elements entirely, opting instead for local element definitions, global types, and no use of anonymous types. OAGIS 8.0 follows the Venetian Blind approach in that it uses no anonymous types in any of its derivations (and discourages OAGIS users from using anonymous types). But because OAGIS 8.0 uses substitution groups to support plug-in extensibility, global elements are used for any OAGIS-extensible element, i.e., for all Nouns, Noun Components, and Components. Even so, OAGIS 8.0 requires that each of these global elements be defined using global types, which preserves most of the benefits of using the Venetian Blind approach.

4.2.2 Blurring of Fields and Compounds

Because XML Schema provides a type system and a rich set of predefined types, it is now possible to represent things like dates, times and the date-time combination using built-in XML Schema data types. This means that instead of using multiple elements to define DateTime it is now possible to use a single element to represent a date, or a time or a date-time combination in ISO 8601 format. Similarly, it is possible to represent integers, decimals, Booleans, floats, etc. (the full set is documented at http://www.w3.org/TR/xmlschema-0/#CreatDt).

Because of this some concepts that OAGIS had previously identified as Segments/Compounds now can be represented as single elements or single

elements with attributes, thereby blurring the previous distinction of fields and segments (now called compounds). For example DateTime, in previous releases has been a segment, now by using the XML Schema dateTime it is a simple element.

4.2.3 "OAGIS-Extensible" Elements

Only Nouns and Components are defined as being ***OAGIS-extensible elements***. As such, each is extensible to carry UserArea extensions and/or overlay extensions. Because this extensibility comes at a cost, Fields and Compounds have not been made OAGIS-Extensible. However, standard XML Schema extensions mechanisms can be used to define new Compound types, based on the existing types.

4.2.4 Typing Components

By converting Component elements into globally usable type definitions, it is possible to ensure consistency across OAGIS.

It is also possible to identify common component types, e.g., Party, Address, etc. which can be defined globally and used throughout OAGIS.

4.2.5 Representation of Nouns

Just as Fields, Compounds, and Components are represented as globally (re)usable types, so are Nouns. By representing Nouns as types, the consistency of each Noun's definition can be achieved across all uses of the noun.

Under OAGIS 8.0 in XML Schema, a single, comprehensive Noun definition is created and used in all relevant BODs. For greatest flexibility and broadest applicability, all parts of the Noun (its Components, Fields, and attributes) are declared as being non-required (minOccurs="0" for elements, use="optional" for attributes). Which of the Noun's Components, Fields, and attributes are required or optional is defined for each BOD in a separate Noun-constraint stylesheet and optionally validated at runtime by applying the stylesheet using any standard XSL processor.

Nouns that are defined in this manner are referred to as "relaxed," in that their structure has been defined but all of the minimum occurrence constraints have been relaxed. In other words, PurchaseOrders have Headers and Lines, but the relaxed definition says that the Headers and Lines are not required to be there. Each OAGIS user can decide whether to define and apply constraining stylesheets (e.g., for the ProcessPurchaseOrder BOD which requires a Header and at least one Line), or she/he can instead validate the correctness within his/her application.

This approach has been taken as an efficient means of ensuring the uniformity of noun content among all uses of the noun. For example, the same PurchaseOrder content is available in the ProcessPurchaseOrder and the GetPurchaseOrder BODs. No special effort is required to ensure this uniformity.

The OAGIS 8.0 Architecture Team has explored and rejected an approach that employs XML Schema *type derivation by restriction*, which fails to ensure uniformity among Noun uses without imposing severe maintenance requirements. In short, the mechanism designed to ensure efficient consistency among uses of a noun is both maintenance-intensive and prone to becoming inconsistent. This is due to the way in which XML Schema's derivation by restriction is performed. In order to define a relaxed model and then subsequently constraint it, one must derive the constrained ("narrowed") noun "by restriction."

Schema's current derivation-by-restriction mechanism would require that a noun's entire structure be replicated and constrained. The unpleasant ramification is that any modifications to the relaxed noun must be manually reapplied to each constrained noun. So, if a "fully constrained" PurchaseOrder – derived from the relaxed PurchaseOrder definition – was defined for ProcessPurchaseOrder (that is, all parts required for processing are set to minOccurs="1") and a "moderately constrained" PurchaseOrder were defined for CancelPurchaseOrder (e.g., the OrderId alone is required), then any changes to the relaxed PurchaseOrder would also have to be applied to the fully constrained and moderately constrained copies. This rapidly becomes unwieldy, especially since, e.g., changes to the abstract type "Order" must also be propagated among all copies of PurchaseOrder, SalesOrder, etc.

4.2.6 Narrowing Nouns

To achieve specificity for a particular use of a Noun, the type can be restricted on a Verb-by-Verb basis through the constraints captured in XPath expressions. Because of this, components e.g., a RequestForQuote or a PurchaseOrder's Header can have a broad, relaxed definition (e.g., many fields, all optional) but have specific uses restricted (or narrowed) by applying the corresponding the XPath-encoded constraints using a standard XSL processor.

4.2.7 Shared Abstract Types

During the course of OAGIS 8.0 build-out, OAGIS architects identified a small but significant number of common abstract types. Establishing these types (e.g., Document and Order) as common base types for several Nouns and/or Components brings a greater degree of uniformity to these types. As such, all subtypes of these base types carry the same content, in the same order with the same attributes and with the same spelling, capitalization, abbreviations, etc., for all content that is shared in common.

Additionally, any change to the base type will be propagated to each of the descendent types, ensuring long-term consistency among similar types. For example, all things that are Orders (PurchaseOrders, SalesOrders, etc.) share a number of fields, compounds, and components in common. When a change to one of these is deemed appropriate to all things of type Order, the change is made to the Order type; thus, all things of type Order are automatically updated.

This approach to modeling abstract types and concrete subtypes augments OAGIS' regular means of component assembly. These complementary approaches (class-subclass and component assembly) together provide the OAGIS user a rich set of solutions to some of the more perplexing and/or maintenance-intensive modeling challenges.

4.3 XML Schema Substitution Groups and OAGIS 8.0 Extensibility

XML Schema, while a significant improvement over DTDs, lacks straightforward support for some key OAGIS features. Chief among them is the ability to conveniently extend Nouns and Components. Solely using XML Schema type derivation by extension would impose a mind-boggling sequence of derivations and namespace manipulations, just to achieve the simple kind of plug-in extensibility that OAGIS users need. Fortunately, Schema does provide another mechanism that can be used to implement OAGIS extensibility: the substitution group.

Using XML Schema substitution groups, it is possible to say that particular elements of the same type (or similar types) can be substituted, one for another, anywhere that the substitution group's "head element" is referenced. For example, if there is a global element Noun (of type Noun) and there are global elements PurchaseOrder, RequestForQuote, and SalesOrder (each being of a type derived from the type Noun), and these new Nouns can be declared to be in the substitution group Noun, then anywhere that the global element Noun is included in a model, a PurchaseOrder, RequestForQuote, or ProductionOrder can be substituted for the Noun element. Moreover, if the Noun element has been declared to be "abstract", then not only can substituting occur, it must occur.[4]

Substitution groups can cross namespaces. This allows a user of OAGIS to extend, for example, the OAGIS PurchaseOrder noun and have their own PurchaseOrder (in their own namespace) be used in its. This forms the basis for the OAGIS 8.0 implementation of overlay extensibility.[5]

To support overlay extensibility for all Nouns, Noun Components, and Components, each must be implemented as a substitution group. To do so, each is defined as a global element, with a type that is a globally defined type. Then, an OAGIS user can extend any OAGIS-extensible element by creating a new, extended element and adding it to the substitution group.

[4] That is, if the element "Noun" appears in a sequence definition and has been declared abstract, it cannot appear in that sequence; only one of its substitute elements can.
[5] Substitution groups do have limitations. An element can participate in one and only one substitution group, making it an inadequate mechanism for representing (meta)classes and their instances (classes). Other restrictions apply. We recommend using this mechanism with caution.

For example, the Noun "PurchaseOrder" is defined as a global element of type "PurchaseOrder." Any OAGIS user can then extend "PurchaseOrder" in the following manner:

1. In a new namespace, "myns", create a global type, based on the OAGIS "PurchaseOrder" type, that extends the OAGIS PurchaseOrderType:

```
<xs:complexType name="PurchaseOrder">
    <xs:complexContent>
        <xs:extension base="oa:PurchaseOrder"/>
    </xs:complexContent>
</xs:complexType>
```

2. Add to this extension any additional element content, e.g., a GrandTotal element:

```
<xs:complexType name="PurchaseOrder">
    <xs:complexContent>
        <xs:extension base="oa:PurchaseOrder">
            <xs:sequence>
                <xs:element name="GrandTotal" minOccurs="0"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

3. Define a global element "PurchaseOrder" in the new namespace, bind it to the new type, and define it as being in the substitution group PurchaseOrder in OAGIS:

```
<xs:element name="PurchaseOrder" type="myns:PurchaseOrder"
            substitutionGroup="oa:PurchaseOrder"/>
```

4. In the xml instance document, define an OAGIS ProcessPurchaseOrder BOD that references the PurchaseOrder from "myns":

```
<ProcessPurchaseOrder
    xmlns="http://www.openapplications.org/oagis"
    xmlns:myns="myNameSpace"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.openapplications.org/oagis
                        ../OAGIS/BODs/ProcessPurchaseOrder.xsd
                    myNameSpace /mynamespace.xsd"
    revision="8.0" environment="Test" lang="en-US">
    <ApplicationArea>
    …
    </ApplicationArea>
    <DataArea>
        <Process acknowledge="Always"/>
```

```
<myns:PurchaseOrder>
    <Header>
    …
    </Header>
    <Line>
    …
    </Line>
    < myns:GrandTotal>1200000.00</ myns:GrandTotal>
  </Process>
 </DataArea>
</ProcessPurchaseOrder>
```

Note that, by using the Substitution Group mechanism, the new, extended "**myns:PurchaseOrder**" can legally use the newly-defined "**myns:GrandTotal**", effectively plug-replacing the OAGIS PurchaseOrder with an extended version.

A substitution-group-based plug-in mechanism is much easier to use and maintain than other XML Schema mechanisms, and the resultant instance documents are both cleaner and more easily understood. The alternative – using typical XML Schema extension mechanisms – is much more cumbersome to use and maintain. Extension of PurchaseOrder, for example, would require the OAGIS user to import the entire ProcessPurchaseOrder BOD and all of its parts, subparts, etc., into "myns" and to subsequently maintain all of those types each time OAGIS is changed. The practical application of this approach rapidly becomes quite confusing, and the OAGI architects felt it to be an unnecessary burden on the OAGIS user.

4.4 XML Schema Namespaces and OAGIS 8.0

Newly introduced in XML Schema (as compared to XML DTDs) – and new in OAGIS 8.0 – is the use of namespaces and namespace prefixes. This section provides an introduction to namespaces and then describes OAGIS 8.0's use of namespaces.

4.4.1 Overview of XML Schema Namespaces

As mentioned above, the ability to define separate ***namespace***s supports the creation and management of separate sets of element names (also, attribute names, type names, etc.), pertinent to various domains of interest, without the risk of name clashes. For example with

separate namespaces one can define a Person's Address and an Address to Congress without creating a name conflict. This is done by creating separate namespaces to define the terms of each domain (e.g., People and Speeches), assigning a ***namespace prefix*** to each (e.g. "person:" and "speech:").[6] These namespace prefixes can then be used to distinguish, e.g. person:Address versus speech:Address. In this manner, separate vocabularies can be combined without risk of name conflicts, e.g, a Person who lives at person:Address can present a speech:Address to Congress:

```
<CongressionalDocket>
        …
        <Event>
                <Presenter>
                        <person:Name> … </person:Name>
                        <person:Address> …</person:Address>
                </Presenter>
                <EventType><speech:Address/></EventType>
        </Event>
        …
</CongressionalDocket>
```

Here, the parser has no confusion about which "Address" is which, because they are from separate namespaces.

In addition, namespaces can build upon other namespaces, with the terms and constructs of some commonly used namespace being referenced by terms and constructs of another. If, for example, a speech:Address was modeled as a richer type, it might refer to the person who presents the speech:Address:

[6] Namespace prefixes are shorthand names that can be substituted for full namespace names, to be used in the Schema documents and XML instances to keep namespace-distinguished names compact and readable. For example, the namespace for XML Schema is "http://www.w3.org/2001/XMLSchema", and the typical namespace prefix is "xs:". It is much more concise to declare an XML Schema element as "xs:element" instead of using the full namespace name {http://www.w3.org/2001/XMLSchema}:element, which is what would be necessary without namespace prefixes.

```
<speech:Address>
        <Presenter>
                <person:Name>Abraham Lincoln</person:Name>
                <person:Address> …</person:Address>
        </Presenter>
        <speech:Text>Four score and seven…</speech:Address>
</speech:Address>
```

In this example, the Speeches namespace builds upon the People namespace by building the element speech:Address using elements person:Name, person:Address, etc., again with no confusion between the two different Address elements (of two discinctly different types).

In this manner, a schema can incorporate concepts from a rich set of domains, building upon existing knowledge and vocabularies. At the same time, the origin of each concept is clear (because each has a namespace prefix), and, for the same reason, the same words with different contextual meanings can be used, side-by-side, unambiguously.

4.4.1.1 Default Namespaces

The readability of an XML Schema can be enhanced by declaring a ***default namespace***, whereby the concepts in one namespace can be referred to without using a namespace prefix. Any concept in a given XML Schema document or XML instance document that has no namespace prefix is assumed to have been defined in the default namespace (or, if no default namespace has been defined, in "no namespace"). In the example above, if the People namespace were declared as the default namespace, the elements Name and Address (without namespace prefixes) would be assumed by parsers to be the ones from the "People" namespace.

Default namespaces can reduce the number of keystrokes required to create an XML Schema document and can reduce the number of characters that flow across the wire in an XML instance document, one that would otherwise be peppered with namespace prefixes. Industry standard practice, though, is to use namespace prefixes for all imported schemas, making it clear which are externally-defined concepts and which are user-defined concepts. Also, by declaring a default

namespace, the user is precluded from using "no namespace" (also referred to as the ***null namespace***), because a parser could not distinguish between elements from the default namespace and elements from no namespace.

4.4.2 OAGIS 8.0 and Namespaces

Like virtually all other XML Schema-based standards, OAGIS 8.0 relies on namespaces to distinguish OAGIS concepts from concepts in other domains. All concepts in the OAGIS standard are defined in the namespace "http://www.openapplications/oagis" (referred to in this document as "the OAGIS namespace").[7] All elements, attributes, types, etc., defined by OAGIS are declared in the OAGIS namespace, making it clear to OAGIS users which concepts are and are not a part of the standard, and, at the same time, protecting their names from clashing with OAGIS concepts.

Those referencing the OAGIS namespace typically use the namespace prefix "oa:" to distinguish OAGIS elements and types from elements and types from other domains.[8]

For example, the type that defines an OAGIS BOD is typically referred to as oa:BusinessObjectDocument; the specific BOD formerly known as ProcessPO is now oa:ProcessPurchaseOrder (given, of course, that the prefix "oa:" has been bound to the namespace "http://www.openapplications/oagis").

4.4.2.1 Off-the-Shelf OAGIS and Namespaces

OAGIS users who use OAGIS without extending it do so by creating a BOD instance document (xml file), pointing it to an OAGIS BOD XML element from the OAGIS namespace. For example, a

[7] While any schema-legal name (xs:NCName) can be used to identify a namespace, OAGIS uses a full URI to identify the namespace (the OAGIS namespace could just as easily have been called "OAGIS"). The use of the schema's URI has become standard practice in identifying namespaces, especially for the namespaces associated with standards such as OAGIS.

[8] The choice of namespace prefix is arbitrary; it can be any lexically valid name. An OAGIS user could use the namespace prefix "o:" or "oagis:" or "OAGIS:" or "TheEBusinessStandard:", so long as their choice of namespace prefixes is legal, and is mapped to the namespace "http://www.openapplications/oagis".

ProcessPurchaseOrder BOD instance points to the relevant XML Schema file, e.g.,

"http://www.openapplications.org/oagis/**ProcessPurchaseOrder.xsd**"

By defining no namespace prefix for the OAGIS namespace, the BOD and all of its numerous parts are treated as being from the OAGIS namespace.

4.4.2.2 Extended OAGIS and Namespaces

Those OAGIS users who want to extend OAGIS have a choice of whether to create their extensions in no namespace or to define their own namespace. Typically, only the "casual OAGIS extender", i.e., one who extends OAGIS only by populating UserAreas, uses the null namespace. In such cases, the OAGIS namespace must be declared along with a namespace prefix; the user's elements and types, being in no namespace, are the only ones that are allowed to be referenced without namespace prefixes.

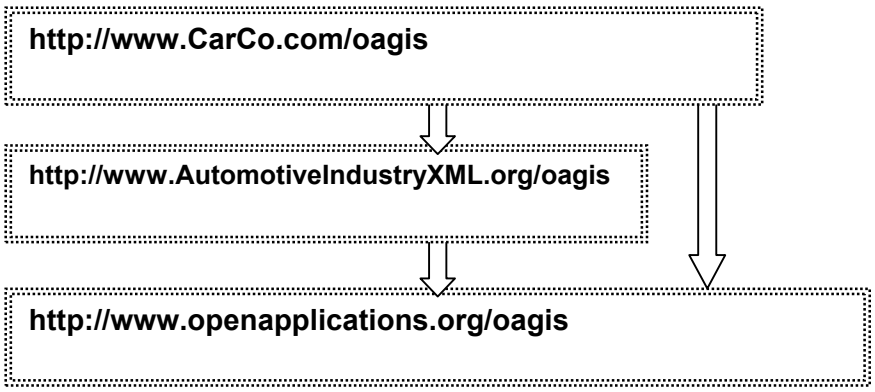
The typical OAGIS extender, though, creates either a specific vertical extension – e .g., for use by the Ford Motor Company in its business interactions, or an industry extension, e.g., across the entire automotive industry. In either case, such extenders are required to each define a separate namespace that defines names relevant to their respective domains. This is the chief means that OAGIS extensions are intended to be built (and built upon).

4.4.2.3 Extended OAGIS Namespace(s) Example

Consider a fictitious organization, the Automotive Industry XML Group and their equally fictitious interchange standard, AutomotiveIndustryXML which captures and encodes automotive industry concepts. The Automotive Industry XML Group could (arbitrarily) assign a namespace, e.g., http://www.AutomotiveIndustryXML.org/oagis to their standard. AutomotiveIndustryXML, being an extension of OAGIS, draws in and builds upon the OAGIS concepts (namespace). Each of the companies affiliated with AutomotiveIndustryXML can use the AutomotiveIndustryXML standard "off the shelf" (i.e., without extending

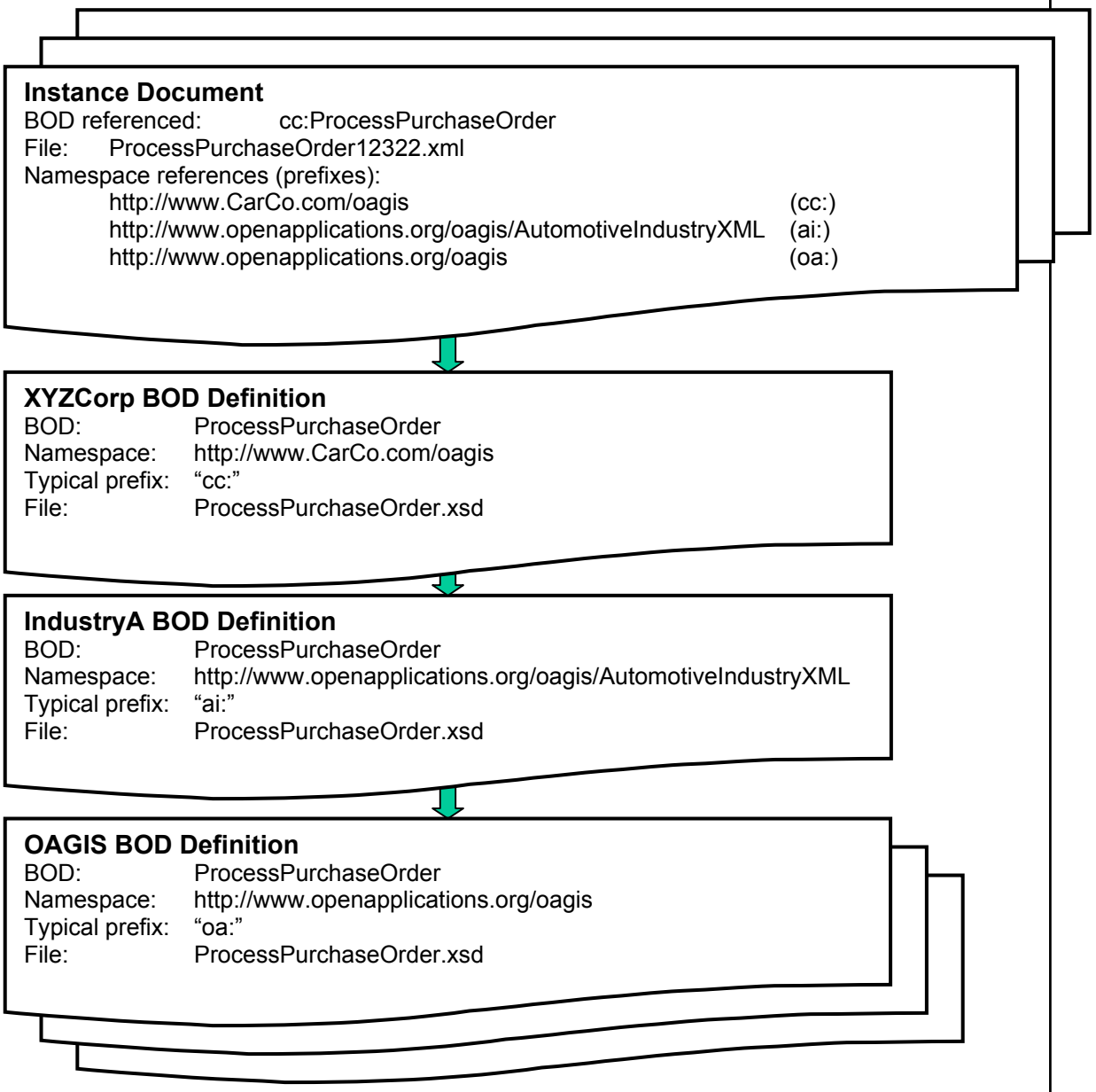
it), or may choose to extend it for use in their own standard. For example, the fictitious company Cc: could extend AutomotiveIndustryXML by adding its own, company-specific extensions to the standard AutomotiveIndustryXML concepts, thus both embracing the AutomotiveIndustryXML standard yet not being totally constrained to AutomotiveIndustryXML's definition of, say, a WarrantyRepairOrder.

The layering of namespaces might look like this:



The AutomotiveIndustryXML namespace builds on concepts imported from OAGIS; the Cc: namespace builds on both AutomotiveIndustryXML and OAGIS concepts (elements, types, etc.). We refer to this as a ***cascade***, whereby concepts from OAGIS cascade to, e.g., an industry extension, and these concepts cascade on to verticals, either extended or not. Prior to OAGIS 8.0, a cascade like this was not possible; creating a single extension to OAGIS was feasible, but was labor-intensive to maintain and modify.

An example of the files involved in the layering might look like this:



The OAGIS BOD definition is built from multiple files in the OAGIS namespace; the file structure and inclusion scheme for the OAGIS files are described Appendix A of this document.

Also note that this is a departure from prior OAGIS versions. In previous versions, the standard means of extending OAGIS was to copy the set of OAGIS files, (oagis_extensions.dtd and oagis_entity_extensions.dtd which referenced your extension dtds) add extensions, share the entire set of files with application builders (for enterprise-internal OAGIS

interchange) and/or with trading partners (for eBusiness transactions), and, once these extensions have been incorporated by the relevant parties, to transact business using that extended version.

OAGIS 8.0 introduces an approach that fits more with the "web model," whereby an authorized version of the standard schema is posted at a particular web site, and those wishing to extend the standard do so in another, distinct namespace. While it is not necessary for systems to be "on the web" to use OAGIS, nor that the OAGIS schema be read remotely if they are on the web, following this approach achieves a more important goal – a pragmatic separation of concerns. With the OAGIS standard encapsulated in a relatively stable namespace, and with extensions clearly segregated and identified as to their purveyor(s), it is clear to the user which standards and extensions they are reliant upon, and is simpler for the separate parties (OAGIS and the extenders) to revise their standards. Using standard XML Schema mechanisms, it is possible for OAGIS, an industry, or a vertical, to upgrade their own content and for that content to be integrated by those incorporating their schema.

4.5 Extensions to OAGIS

OAGIS 8.0 preserves and modifies the existing form of extensibility – USERAREA extensibility – and introduces a new kind of extensibility, "Overlay" extensibility.

A key feature of the OAGIS standard is its extensibility. It has long been the experience of OAGIS users that, no matter how forward-looking a standard might be, the standard's creators cannot foresee all possible uses and therefore all required content. A standard that is not extensible can hamstring its users by preventing them from including necessary content. They end up either making do until the next release cycle, or, worse, they end up "temporarily" stuffing data values where they don't belong. A widely used alternative is to create a very general schema that is very open, yet offers little structure or guidance, and virtually no validation of the content. Such an approach places a heavy burden on the application developer to interpret and validate the content.

A standard that is extensible provides the user with the ability to add information that would otherwise not fit in the standard, either for temporary use until incorporated into a future release, or for limited-scope use (in cases where the extended information is of little interest/value to the full set of the standard's users).

Either way, extensions do represent a variance from the standard, and all participants that use the extended standard must recognize the extensions. This compromise reduces the set of negotiated interchange items to the relatively small set of extended items; the bulk of the standard continues to represent a stable interchange contract.

Still, any extensibility scheme must also support the users of an extended standard in their interchange and maintenance of extensions. If an extension is embedded in the files of a copy of the standard, it will be difficult for partners to accommodate and will make OAGIS version updates labor-intensive to install. The mechanisms supporting OAGIS 8.0 extensibility have been designed not only with extensibility in mind, but also with the goals that users' extensions 1) will be easily sharable with interchange partners and 2) will accommodate OAGIS version upgrades without significant additional labor.

4.5.1 Design Goals for OAGIS Extensibility

OAGIS extension mechanisms are somewhat complex in their inner workings, but were designed with these principles in mind:

- User-defined extensions to OAGIS should neither require nor depend on user modifications to any of the OAGIS Core xsd files. To ease the transition between OAGIS releases, OAGIS Core xsd files should be kept separate from user extension xsd files. In as much as possible, the OAGIS/Schema user should be able to easily replace the OAGIS core files without teasing apart any intertwined extension definitions, and with no discernible user impact (save for the unavoidable impacts of changes to extended content).

- Regardless of the complexity of the underlying extension mechanisms, the user should be able to follow a set of simple,

repeatable instructions for extending OAGIS, and should not have to go through the painful exercise of determining which OAGIS files to include, and in which order.

- Any extension of OAGIS must occur in a namespace other than OAGIS. This allows the clear delineation of ownership between the specification and any extensions. This preserves the separation of the core standard from nonstandard or quasi-standard extensions, clarifying to the OAGIS user the boundary between core and extended content. This also helps OAGIS maintainers and extenders in their management of change, knowing that changes to OAGIS content will suffer no name clashes with their extended content.

4.5.2 UserArea Extensibility

OAGIS versions prior to OAGIS 8.0 offered extensibility via the user area mechanism, whereby extended content could be added in segregated USERAREA elements, as child elements under the USERAREA element:

```
<USERAREA>
    <GTOTAL>12345.00</GTOTAL>
</USERAREA>
```

This practice is still supported in OAGIS 8.0, in the form of the new ***UserArea*** element. Like its predecessor, the new UserArea extensibility supports the extension of OAGIS by adding any user content (just so long as it represents legal XML). One key difference is that any new concepts must be defined in a separate namespace. A UserArea can contain OAGIS components from other BODs that may be thought to be relevant to the BOD being extended. To include these, no special treatment is needed:

```
…
<UserArea>
    <SalesPerson>…</SalesPerson>
<UserArea>
…
```

However, if an OAGIS user also adds components that have not been defined in the OAGIS namespace, they must be defined in a namespace other than the OAGIS namespace, for example:

```
…
<UserArea>
    <SalesPerson>…</SalesPerson>
    <ai:VehicleWarrantyExpirationDate>
    …
    </ai:VehicleWarrantyExpirationDate>
<UserArea>
…
```

A key benefit of these namespace prefixes is that they clearly distinguish extensions from core concepts.

OAGIS 8.0 UserAreas typically appear as the last element in any OAGIS component.[9] Their use is entirely optional. If used, the UserArea can contain any number of syntactically correct XML elements, *so long as each element has been defined in some schema;* the OAGIS Schema, a referenced industry and/or vertical schema, or in a schema defined and referenced by the user.

With UserArea extensibility, it has been possible to extend OAGIS on the fly, without having to wait for the release and distribution of a subsequent OAGIS version that incorporates the extension.

4.5.3 Overlay Extensibility

A drawback of UserArea extensibility is that it relegates extended content to being subordinate to the UserArea element – content that could otherwise be regarded as having the same level of importance as existing OAGIS content. OAGIS 8 recognizes the needs of industries and verticals to extend the schema with elements that are legitimate peers of OAGIS elements; that is, with extended content that is not forced to be segregated under a UserArea. This type of extensibility has been referred to as ***overlay extensibility***, in that the extended content appears *in line with* the OAGIS content. Using overlay extensibility, a GrandTotal element

[9] Always as the last element of the core OAGIS definition, but before any extended content that may occur as an Overlay.

can appear at the same level as other content of equivalent importance, not relegated to being subordinate to the UserArea:

```
…
<Item>…</Item>
<Item>…</Item>
<Item>…</Item>
<GrandTotal>12345.00</GrandTotal>
…
```

In OAGIS 8.0, by design, all Nouns, Noun Components, and global/shared Components are *overlay extensible*. That is, any OAGIS Noun, any of its components, and any globally reusable OAGIS Component can be extended, appending new, namespace-prefixed content at the same level of element nesting as other element content.

For reasons of both "clean modeling" and, more importantly, tractable and efficient parsing, XML Schema requires that all extensions appear as the last parts of a particular element's or type's content. So any extended content will appear as appended content, with OAGIS-defined content appearing first.

To emphasize the distinction between OAGIS core content and extended content, namespace prefixes are also required for all extended content. The result is that standard OAGIS content appears first (with or without the OAGIS namespace prefix, depending on the user's preference) followed by the content added by the first extender, followed by the content added by the second, and so forth.

So, in the case of a Ford extension of the AutomotiveIndustryXML extension of the OAGIS ProcessPurchaseOrder BOD, the content of the OAGIS PurchaseOrder might look like:

```
<PurchaseOrder>
    <Header>…</Header>
    <Line>…</Line>
</PurchaseOrder>
```

An AutomotiveIndustryXML extension that adds the GrandTotal element to the PurchaseOrder might look like this:

```
<ai:PurchaseOrder>
    <Header>…</Header>
    <Line>…</Line>
    <ai:GrandTotal currency="USD">43000.00</ai:GrandTotal>
</ai:PurchaseOrder>
```

meaning that a AutomotiveIndustryXML Purchase Order (as depicted by the "ai:" prefix) contains a GrandTotal element (also defined in the AutomotiveIndustryXML namespace), that has the value 43000.00 US Dollars.

Now, if a company CarCo wanted to call out, for example, the total destination charges in a PurchaseOrder, CarCo's extension to the AutomotiveIndustryXML PurchaseOrder might look like this:

```
<cc:PurchaseOrder>
    <Header>…</Header>
    <Line>…</Line>
    <ai:GrandTotal currency="USD">43000.00</ai:GrandTotal>
    <cc:TotalDestinationCharges
            currency="USD">2150.00</cc:TotalDestinationCharges>
</cc:PurchaseOrder>
```

Note that the CarCo PurchaseOrder also includes AutomotiveIndustryXML's GrandTotal element, because the Ford PurchaseOrder extends the AutomotiveIndustryXML PurhcaseOrder.

Also note that, since Header and Line came from OAGIS and GrandTotal came from AutomotiveIndustryXML – both higher up in the cascade – these elements appear *above the CarCo-specific extensions*.[10] While this might not look exactly how CarCo might like to see it (i.e., they'd want the GrandTotal element at the end), remember that this is an XML instance document, used by applications and almost never seen by people. The people, on the other hand, would see a made-for-humans presentation (e.g., as transformed by a stylesheet) – such a presentation would certainly need to accommodate the viewers' preferred presentation ordering; XML instance documents need not.

[10] The way that OAGIS overlay extension is implemented guarantees that the schema will enforce this ordering.

Note: OAGI recommends that applications sending and receiving XML content not be reliant on the ordering of the elements (fields, compounds, and components) within a given XML element. While it is important to be aware of the representation of an XML document and it's defining XML Schema, the sequence in which these elements occur at a given level should not be forced back to the business application. Granted the XML parser must place the content in the correct sequence (according to the corresponding XML Schema document) the business application should not be forced to duplicate this sequence. This is especially important in the eventuality that a future XML Schema Recommendation would support a less constrained ordering.

OAGIS Overlay extensibility is made possible by the use of a substitution-group plug-in design pattern. Each Overlay-extensible OAGIS element is implemented as an XML Schema substitution group, allowing for each element to be plug-in-replaced by an extended element from another namespace.

4.5.4 Enumerations Inextensibility

Throughout OAGIS there are examples of attributes and elements that have values drawn from a fixed, limited set of values. Simple examples might include an attribute of the type DaysOfTheWeek, with legal values Monday, Tuesday,… Sunday. We refer to these as enumeration types. These are implemented in XML Schema by defining, for example, a string type and restricting it by enumerating the legal values:

```
<xs:simpleType name="DaysOfTheWeek>
        <xs:restriction base="xs:string">
                <xs:enumeration value="Monday"/>
                <xs:enumeration value="Tuesday"/>
                <xs:enumeration value="Wednesday"/>
                <xs:enumeration value="Thursday"/>
                <xs:enumeration value="Friday"/>
                <xs:enumeration value="Saturday"/>
                <xs:enumeration value="Sunday"/>
        </xs:restriction>
</xs:simpleType>
```

While OAGI endeavors to use enumeration types for only the most stable of situations (i.e., where the enumerations are fixed and unlikely to change), there are times when a particular enumeration might not have been foreseen; and the use of the unanticipated value would be rejected by any validating parser.

Say, for example, that the OAGIS schema were to rely on simple type value enumeration to validate the correct use of international currency codes, which are, in general fairly stable. If a new currency were to appear, due to the formation of a new country or the restructuring of an existing country's monetary policy, the new currency code would have to be accommodated, until such time that the OAGIS standard could incorporate the new code.

But no company should be prohibited from doing business in that currency, just because a validating parser rejected its currency as being unrecognized. Without some means of addressing this change the only options for the OAGIS user would be to copy and edit the OAGIS schema, (rendering it incompatible with other users' understanding of the standard), or to use the value and disable validation of the instance documents (requiring all who receive the data to do the same). Neither is an acceptable solution.

Unfortunately, the XML Schema Recommendation, at the time of this writing, supports no method of extending a set of enumerated values, short of editing one or more files in the defining schema. This harsh constraint has been explained as being necessary for the efficient processing of enumerations. And it has profound ramifications on the OAGIS design.

OAGIS 8.0 does provide a way to partially address this situation: the use of the "semantically named element" design pattern, which provides an extensible alternative.

Clearly, the best solution is to avoid the use of enumeration types for all but the most stable of value sets (e.g., Yes/No, days of the week, etc.) and to find alternative means of validating volatile content, either by using more extensible constructs or by relegating validation to the application.

4.6 Value-sets and Validation

XML Schema, as a typed language, provides the capability of checking aspects of both the form and the content of the XML instance document – to a much greater extent than was possible with DTDs. And the first reaction of seasoned XML DTD users is typically to "nail down" everything in sight, given the new hammer that they've just discovered. In many cases, this allows the language that they're building to be more precise and the XML instance more reliable than before. But in many cases, the language that they've created can be inflexible in the face of expected change.

In prior sections we discussed the inextensibility, in XML Schema, of enumerated value sets; that any set of values that is represented as *XML Schema simple type value enumerations* could only be expanded by altering the original source file where the enumerations are declared.

In general, the representation in a schema of any set of values needs to be carefully considered, especially if there is a strong desire to check the validity of these values using a schema-validating parser. A solution that is too open may place too much of a burden on the application to validate that the values are correct; a solution that is too strict may result in a show-stopping and irreparable rejection of the XML instance by the validating parser.

How value-sets are represented (and validated) is directly related to the volatility of the value-set, which can be rough characterized as:

1. Totally stable – the values in the value-set are fixed, either by nature or by decree, with (statistically) no possibility of change. Examples include "YesNoAnswer" (with values "Yes" or "No", but no others), "USDaysOfTheWeek" (Monday…Sunday), and "ThreeMajorStatesOfMatter" (Solid, Liquid, Gaseous).

2. Mostly stable – the values in the value-set are believed to be stable, but there is a rare possibility that new values could be needed, either at predictable or unpredictable times. Examples include CountryCodes or CurrencyCodes, which are mostly stable, but can change as the international political and monetary landscapes evolve.

3. Somewhat fluid - the values in the value-set are fairly stable, but there is the expectation that new values will be needed, either at predictable or unpredictable times. Examples include Party types (ShipTo, BillTo, Carrier, etc.), ChargeCodes (Transport Costs, Basic Freight, etc.)[11]

4. Highly volatile – the values in the value-set are, by nature, likely to change, either by the frequent and unpredictable addition, replacement, or removal of values, or because the values are highly context sensitive and thus subject to change in different business circumstances. Examples include ProductCodes, CatalogItemNumbers, etc.

On the one extreme, only completely stable values sets should be considered for representation using XML Schema simple type value enumeration. In any other case, the need to extend the value-set simply couldn't be accommodated in a timely manner, and, until the next OAGIS schema release, a validating parser will reject any new value. This is because of the absolute prohibition in XML Schema of simple type extension (and, thus, the inextensibility of simpleType value enumerations).

On the other extreme, those value-sets that are known to be highly volatile should be validated by the application(s), after a suitable Sync operation has been performed. No schema-level enforcement should be relied upon, due to the fluidity of the language that these values represent.

Other value-sets are likely to have their validation deferred to the application. This requires that a Sync operation occur in order to accommodate the synchronization of this application check.

Note: That it may be possible to do some of these checks outside of the application by extending the constraints provided by OAGIS 8.0. Future releases of OAGIS may also provide assistance in this area.

[11] Note that in some standards, e.g., EDIFACT Charge Types, somewhat fluid and mostly stable valuesets can be driven toward highly stable by the establishment of a standard set of values. However, this approach often contains a widely used escape such as "Miscellaneous" to cover for unanticipated cases; without a means of capturing a specific other values, the meaning and utility of such valuesets rapidly diminishes. OAGIS encourages the use, when needed, of the more extensible models to capture and validate the appropriate spectrum of values.

However, there is a special yet common case where the value-set could be represented using the OAGIS ***semantically named element*** design pattern and validated by the parser. This particular approach to value-set extensibility is covered in the next section.

4.7 Semantically-Named Element Sets

This design pattern is most appropriate where enumerations have been used to describe the "type" of a thing, e.g., a Party Type (ShipTo, BillTo, Carrier). In many standards, separate types are often combined into a single, generic type, and each instance is distinguished by a "type" attribute. So, for example, rather than creating a specific type for each kind of Party, a generic Party element is defined and its "type" is represented only in an enumeration (type="ShipTo").[12]

Given that new Party types are expected to be added by OAGIS extenders, and given the fact that there is not a convenient way to extend an enumeration in XML Schema, the OAGIS design team was faced with a choice: either have such things as Party Type not be validated, or find a means of extending OAGIS such that new Party types could be added by the OAGIS extender and subsequently validated. OAGI has opted for the latter.

In cases where a "type" attribute exists and is likely to need extending, these types are instead represented as proper XML Schema elements – elements that are of the same or a closely related (derived) type. Their value-sets, rather than being enumerated sets of string values, are captured in XML Schema Substitution Groups. Finally, these elements are wrapped in a pair of tags named for the original (more abstract) type.

So, in the case of Party type, rather than having N tags named "Party", each having a type="*some party type*":

```
<Party type="Customer" active="false" oneTime="false">
        <Name>…</Name>
        <Currency>USD</Currency>
        <Address>…</Address>
        <Contact>…</Contact>
</Party>
<Party type="Supplier" active="false" oneTime="false">
```

[12] This led to a relatively bloated generic type: the properties of any specific party type were added to the generic.

```
            <Name>…</Name>
            <Currency>USD</Currency>
            <Address>…</Address>
            <Contact>…</Contact>
</Party>
<Party type="Carrier" active="false" oneTime="false">
            <Name>…</Name>
            <Currency>USD</Currency>
            <Address>…</Address>
            <Contact>…</Contact>
</Party>
```

there will now be a set of Parties, with the type of each captured in the tag name:

```
<Parties>
   <CustomerParty active="false" oneTime="false">
      <Name>…</Name>
      <Currency>USD</Currency>
      <Address>…</Address>
      <Contact>…</Contact>
   </CustomerParty>
   <SupplierParty active="false" oneTime="false">
      <Name>…</Name>
      <Currency>USD</Currency>
      <Address>…</Address>
      <Contact>…</Contact>
   </SupplierParty>
   <CarrierParty active="false" oneTime="false">
      <Name>…</Name>
      <Currency>USD</Currency>
      <Address>…</Address>
      <Contact>…</Contact>
   </CarrierParty>
</Parties>
```

While this may seem odd at first, it has one key benefit:

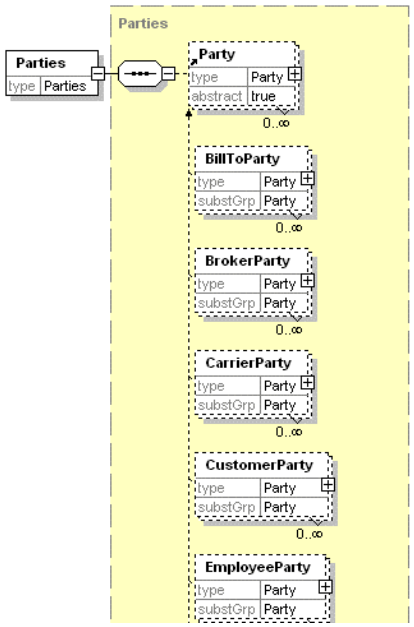
- It is extensible and validatable under XML Schema – the prior way is not.

Also, unlike its inextensible counterpart, the Substitution Group implementation renders all group members accessible anywhere that the group has been referenced. So, if an OAGIS extender such as AutomotiveIndustryXML were to add the new party element "ai:DealershipParty", that new party element would be available in all OAGIS BODs used by AutomotiveIndustryXML, without any additional work being done.

To distinguish these semantically-named elements from other global elements, their name takes on the suffix of their schema type definition (you may have noticed that in ShipToParty, BillToParty, etc.).

To simplify application processing, all things in the substitution group Party are grouped together and captured under the "Parties" element. This simplifies the processing of a set of party-type things (//Parties/*), and has the added benefit of reducing the element bloat of the parent element.

In a semantically-named element set, there is a grouping element, Parties, which contains 0 or more elements in the substitution group Party:



The "Party" element, of type Party, is declared "abstract" and must be substituted for, by any member of the substitution group "Party". So the diagram above depicts that any member of the Party substitution group (BillToParty, BrokerParty, etc.) may appear any number of times under the Parties element. As substitution group members, each must be of the same type (the type Party) as the substitution group's head element (the Party element) ***or a valid derivation*** of that type. All of the members depicted above are of the type "Party", so none adds anything special to extend the Party type, which might defined as:

```
<xs:complexType name="Party">
    <xs:sequence>
        <xs:element name="Name" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="PartyIds" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="Currency" type="Currency" minOccurs="0"/>
```

```
            <xs:element name="Description" type="Description" minOccurs="0"
                        maxOccurs="unbounded"/>
            <xs:element name="GLEntitySource" type="GLEntity" minOccurs="0"/>
            <xs:element name="PaymentMethod" type="PaymentMethod" minOccurs="0"/>
            <xs:element name="Rating" type="Rating" minOccurs="0"/>
            <xs:element name="TaxExempt" type="xs:boolean" minOccurs="0"/>
            <xs:element name="TaxId" type="Id" minOccurs="0"/>
            <xs:element name="TermId" type="Id" minOccurs="0"/>
            <xs:element ref="Address" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="Contact" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="Attachment" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="UserArea" type="UserArea" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="active" type="xs:boolean" use="optional" default="false"/>
        <xs:attribute name="oneTime" type="xs:boolean" use="optional" default="false"/>
    </xs:complexType>
```

resulting in an XML instance that would look like this:

```
<Party active="…" onetime="…">
        <Name>…<Name/>
        <PartyIds/>
        <Currency/>
        <Description/>
        …
        <Contact/>
        <Attachment/>
        <UserArea/>
</Party>
```

It would be possible – even desirable – to create specialized schema types to accommodate appropriate content. For example, a ShipToParty type could be created as an extension of the Party type, adding the element "LoadingDockHours" – relevant to the ShipToParty but not likely relevant to other Party types. So the type ShipToParty could be derived by extension from type Party, adding a LoadingDockHours field (of type TimePeriod):

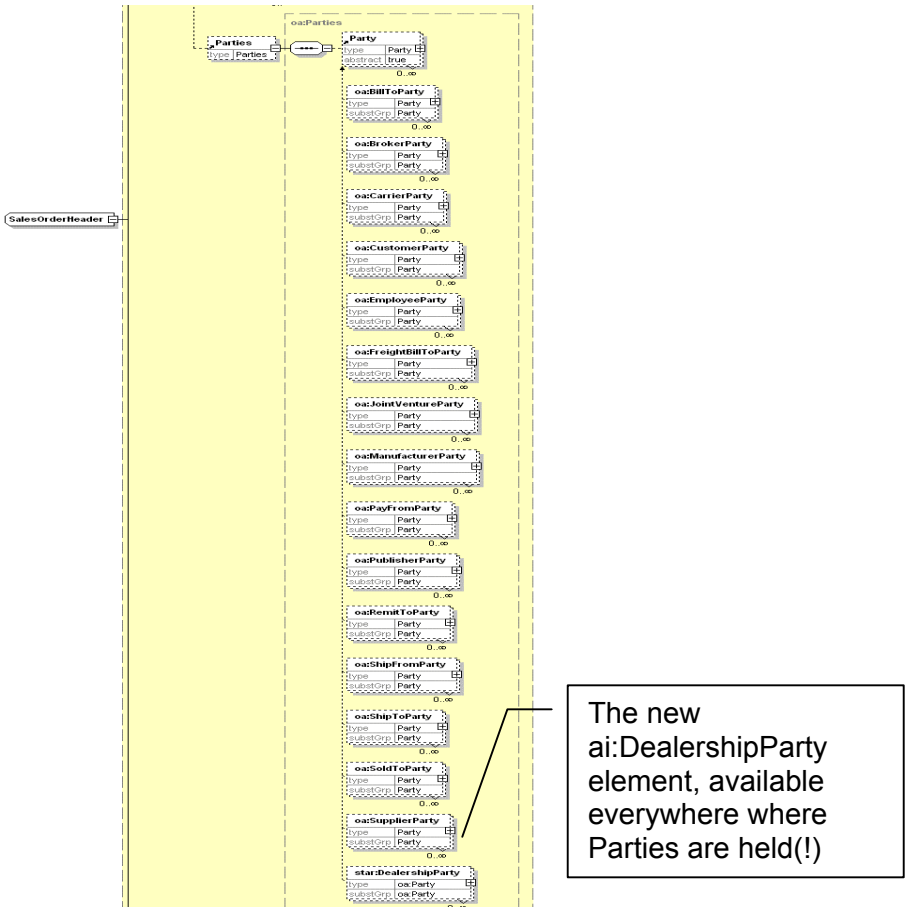
```
<xs:complexType name="ShipToParty">
    <xs:complexContent>
        <xs:extension base="Party">
            <xs:sequence>
                <xs:element name="LoadingDockHours" type="TimePeriod"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Rather than adding these specific elements to the general Party type the generic Party type remains simple, containing only those properties

common to all Parties; each specific party would have its own specific properties (e.g., ShipToParty type with element LoadingDockHours).[13]

Finally, any OAGIS extender can add a new Party element of type Party (or some legal derivation), add it to the substitution group Party, and it would be available anywhere where Parties appear:

```
<xs:element name="DealershipParty" type="oa:Party"
        substitutionGroup="oa:Party"/>
```



One final note: OAGIS employs a grouping element (not to be confused with XML Schema groups) to collect semantic elements of the same base type but with different names ("Parties" collects, e.g., ShipToParty, SupplierParty, etc.). As discussed above, this is done to provide a hint to applications as to which of the extended items are related and can be

[13] Even though these may all be represented in a database table that recombines all of the distinguished properties into a composite record, the data that goes into the table will be appropriately differentiated – you won't find a LoadingDockHours in a PublisherParty record, because the XML won't allow it.

processed together. However, OAGIS recommends against using such grouping elements for other elements, e.g., for repeating of the same type and same name, such as "Items" grouping repeating "Item" elements. This is thought to be an unnecessary redundancy, since all such items can be retrieved using the element name ("Item") and thus do not need the extra structure (Items/Item).

4.8 File Organization

OAGIS 8 BODs rely upon several resource files. These files are divided in to directories that indicate the purpose of the files that they contain. This design of the file structure arises from a number of design goals and constraints.

First, the files are segregated into categories of OAGIS-defined content and user-defined content. The directories and files are organized both to separate various concerns/concepts and to facilitate the various OAGIS 8.0/Schema extension mechanisms. These mechanisms, and their particular implementation in XML Schema, dictate that some files be kept separate, even if it appears that some might benefit from being combined.

None of the OAGIS files should be modified by the OAGIS user. Doing so will almost certainly render that deployment incompatible with other OAGIS installations, and will make OAGIS version updates labor-intensive to deploy. Extensions to OAGIS are made in separate directories and files. See the subdirectories under the "OverlayExamples" directory for examples of how to craft overlay extensions.

Appendix A in this document provides a list of all the directories and files used in OAGIS 8.0.

4.9 Self-Documenting Schemas

Documentation that is collocated with an XML Schema document is more accessible to the developers who need it, and much easier to keep consistent with the schema. All fields, compounds and types make use of the annotation capabilities of XML Schema to document their usage.

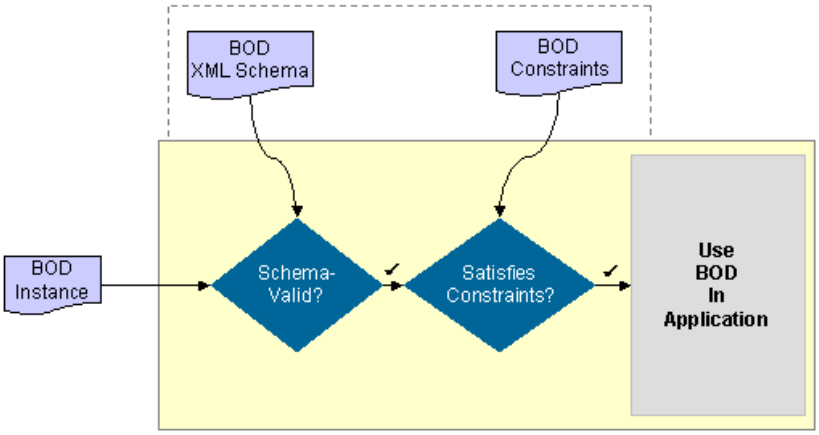
OAGIS documentation has been generated directly from the annotations within the XML Schema. The separately maintained Microsoft Word-formatted OAGIS Documentation has been replaced by generated, html documentation.

4.10 Validation Beyond XML Schema Validation

DTDs are known to provide limited definition of – and validation of – the structure and content of an XML document. All information that cannot have its integrity validated using DTDs must instead be validated by the application; this is true of most OAGIS deployments prior to OAGIS 8. While some OAGIS users prefer this, it does place a heavy burden on application builders to understand and interpret the integrity rules for each BOD and to track these rules as they change. Chances are good that different vendors' applications will interpret these constraints differently.

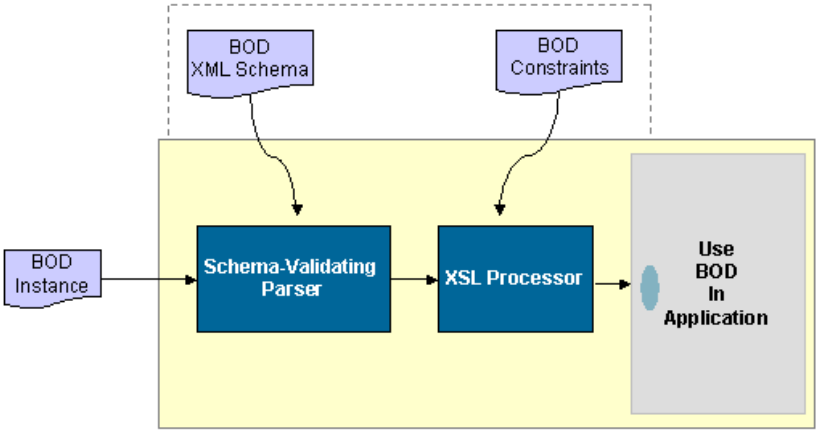
In spite of the progress that XML Schema represents, a Schema-validating parser still only performs limited structural validation. As such, some types of BOD integrity cannot be checked by a schema-validating parser, so other means of integrity checking are often necessary.

In OAGIS 8, it is still possible (and for some data, desirable) to have the application validate the integrity of the BOD data. OAGIS BODs can also be checked outside of the application using a combination of the BOD's XML Schema documents (for structural validation) and OAGI-provided integrity constraints (for most other validation):



BOD Validation – Simplified View

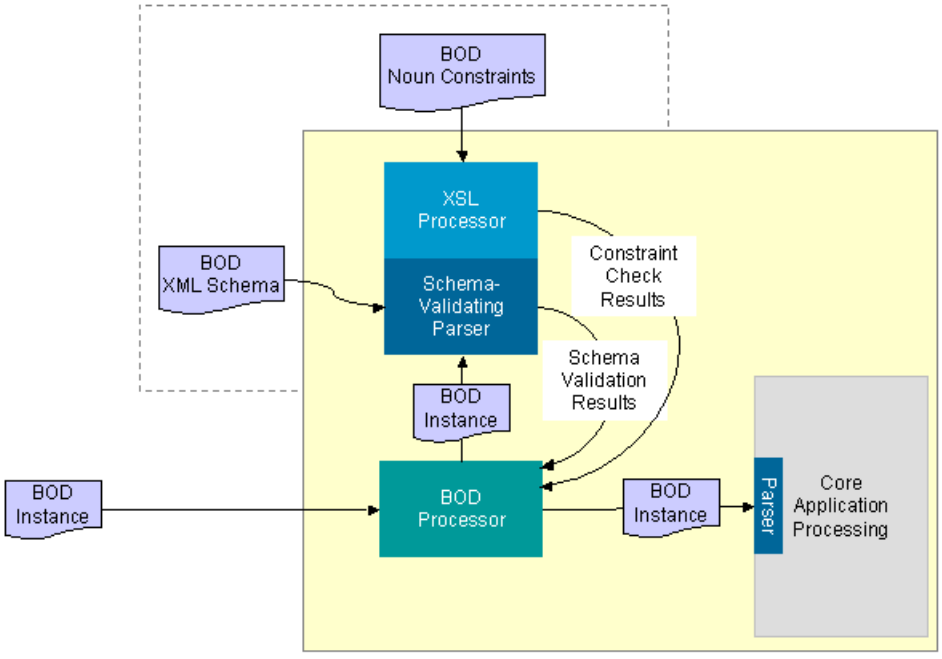
This is accomplished using nothing more than a standard Schema-validating parser, a standard XSL processor and a stylesheet:



BOD Validation Using Standard XML Technologies

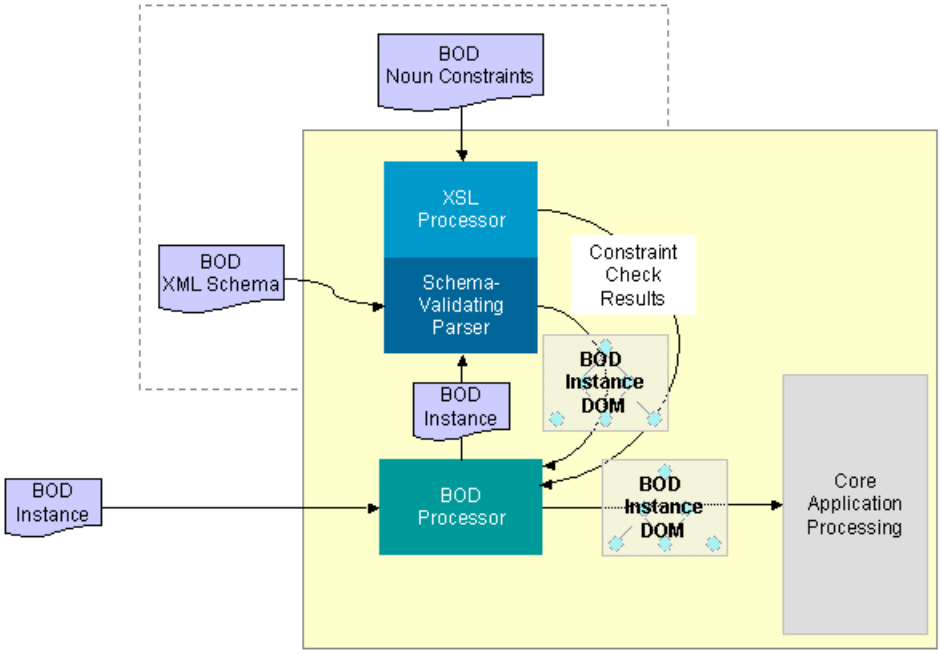
Note that even with this pre-validation of the BOD, it may still be desirable for the application to perform some of its own integrity checks. However, routine checking need not be duplicated in the application.

One possible architecture for this checking involves the use of a standard schema-validating parser, coupled with a standard XSL processor, which jointly validate the structure and semantics of the BOD:



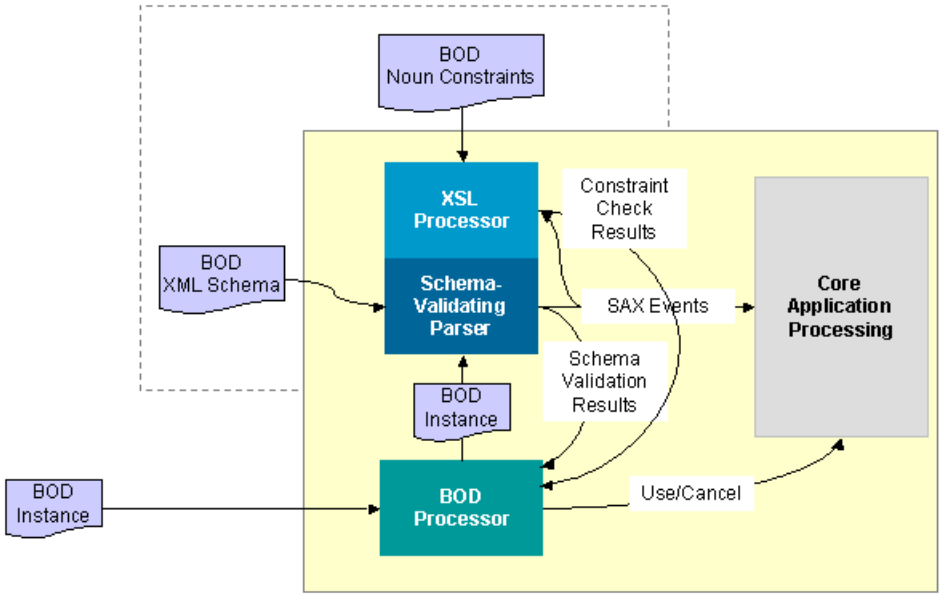
Upon successful validation, the BOD can then be handed off to the application for re-parsing (not validated this time) and use. For typical BODs, redundant parsing will not impose significant overhead.

For greater efficiency, a DOM-based parser and XSL processor can be used:



In this case, the parser generates a DOM that, upon successful Schema validation and integrity rule checking can be handed off to the application, which can use the BOD's DOM form without reparsing.

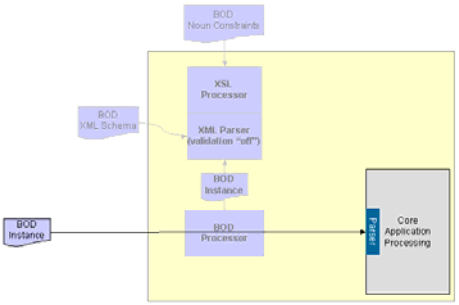
In some cases, though, some BODs are too large to be efficiently re-parsed or to be represented in an in-memory DOM. In such cases, it is likely that a SAX-based approach will be necessary:

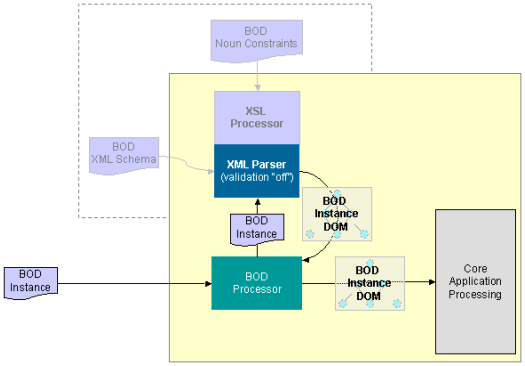


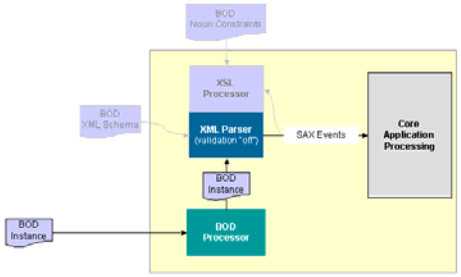
BOD Validation – SAX-event-based (large BODs)

In this case, a SAX-based Schema-validating parser will parse and validate the BOD against it respective schema document(s). At the same time, both the SAX-based XSL processor and the application will listen to the SAX events; the XSL processor will apply the BOD integrity rules, and the application will respond in its own fashion to the SAX events. However, it is only when the parser and the XSL processor report success that the application can commit to acting on the information collected during the processing.

Regardless of which architecture(s) are chosen, it is possible for the application to be run without any external validation of the BOD instance – that is, with external validation disabled:







This comes at some risk, though, especially if the applications that interchange the BODs can change and/or when the BOD(s) are interchanged with an

untrusted interchange partner. Operation will be more streamlined, but at the risk of receiving incorrect and/or incomplete BOD data.

4.11 Parser & Tool Compatibility

All XML Schema documents and XML instance examples have been validated against the following XML tools (in order from development to run-time):

1) Xerces 2.0.1 (jars dated 4/5/02 or later), Apache (a production parser)
2) MSXML 4.0 (SP1 or later), Microsoft (a production parser)
3) XML Spy, v 4.3, Altova (an IDE)
4) XSV v 1.4, Henry Thompson (a development-time validation tool)[14]

Additional parsers and IDE's to be tested are:

1) TurboXML 2.3, TIBCO (an IDE)
2) Schema Quality Checker, IBM (a development-time validation tool)
3) Oracle XML Parser, Oracle (a production parser)

OAGI has worked with Apache (Xerces), Microsoft (MSXML), Altova (XMLSpy), and Henry Thompson (XSV), to correct errors in each respective party's tools – errors that might have impeded progress in deploying a fully usable OAGIS 8. In addition, OAGIS architects have actively participated in the XML Schema Dev listserve, to determine the correct interpretation of some of Schema's capabilities.

OAGIS users who encounter issues with the correct validation of OAGIS models using the aforementioned XML tools may contact OAGI for guidance in getting these issues resolved.

[14] Henry Thompson's XSV is widely regarded as a key reference implementation for XML Schema validation, and is a well-exercised interpretation of the XML Schema Recommendation's syntactic and semantic constraints.

5.0 SUMMARY

While this document identifies the issues with OAGIS prior to version 8, and identifies how these issues will be resolved in OAGIS 8, it is important to remember the things that OAGIS does right:

1) OAGIS is extensible and we are expanding on that extensibility as we move forward.

2) OAGIS is easy to read and understand both at the documentation level and the DTD level. XML Schema itself is not quite as easy to read, however we are making strides to retain this aspect of OAGIS even in the XML Schema instantiation.

3) OAGIS is, and will continue to be, focused on addressing the needs of the business analyst.

4) OAGIS has provided means of abstracting away much of the complexity inherent in the underlying representation language (DTDs), and endeavors to do the same in its XML Schema incarnation.

While the XML Schema project has been on the OAGI radar since 1999, XML Schema has only been released as a W3C recommendation as of May 02, 2001. Even then, it was unclear how all of the XML Schema constructs would be interpreted by validating parsers, and equally unclear how the various OAGIS extensibility mechanisms were to be implemented under Schema. As tool support for, and expert knowledge of, the detailed semantics of XML Schema increases, so does the OAGI's ability to support OAGIS in a manner that addresses our design principles and meets our objectives.

APPENDIX A – OAGIS 8 XSD FILES & DIRECTORIES

This appendix provides a complete list and description of the files and directories to be delivered with OAGIS 8.0 in XML Schema. (Subject to change between now and final release).

1) **OAGIS** (directory)**–** the main OAGIS directory which contains the following. **These files/directories are not to be modified.**

   a) **Resources** (directory)– The OAGIS resource files that are used to build the BOD Instance Schemas. The user must not modify these files.

      i) **Nouns** (directory) – contains the individual noun files that define each of the OAGIS Nouns. One of these files is used in each BOD instance.

      ii) **Verbs** (directory) – the individual verb files that define each of the OAGIS Verbs. One of these files is used in each BOD instance.

      iii) **Components.xsd** – contains the type definitions for each of the common Components and the instantiation of its corresponding global element (needed for component extensibility).

      iv) **Enums.xsd** – contains the OAGIS defined enumerated lists or value lists.

      v) **Fields.xsd** – contains the type definitions of OAGIS Fields and Compounds.

      vi) **Meta.xsd** – some of the meta-information that define core OAGIS concepts.

   b) **BODs** (directory) – Contains the individual OAGIS BODs as defined in XML Schema and the correspond constraints defined in XSL. These are the files that the OAGIS user's XML instance files point to as the defining (and validating) BOD schema. The user must not modify these files.

   c) **BODConstraints** (directory) – contains the support for applying the XPath-based constraint expressions that are used to augment XML Schema validation.

i) **Rules -** (directory) – contains a file for each BOD that contains the constraint rules in XPath for each BOD as defined in OAGIS.

ii) **Generated -** (directory) – contains an xsl file that applies the rules to the BOD instance. These xsl files are generated by form the Rules above.

d) **BODExamples** (directory) – Examples of OAGIS BOD instances (.xml files). These files (and this directory) are not required for OAGIS to function properly.

2) **Documentation** (directory) – contains the documentation for OAGIS in html format. This directory contains both html and other directories that provide the OAGIS 8.0 documentation.

3) **Overlay Examples** (directory) – contains examples of the directories and files that an OAGIS user might create/modify in order to extend OAGIS.

4) **Utility** (directory) – contains software and scripts used for checking, validating, parsing, etc., BOD instance (xml) files and user overlays. (See the ReadMe file for more information)**.**

.